

Programmavariatie voor softwarebeveiliging

Program Variation for Software Security

Bart Coppens

Promotoren: prof. dr. ir. K. De Bosschere, prof. dr. ir. B. De Sutter
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. J. Van Campenhout
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2012 - 2013



ISBN 978-90-8578-603-0
NUR 980
Wettelijk depot: D/2013/10.500/36

For my family and friends.

"You just keep on trying till you run out of cake."
— GLaDOS. From Jonathan Coulton's *Still Alive*.

Dankwoord

De weg naar een doctoraat is lang, en veel mensen hebben mij, al dan niet indirect, geholpen. Het is helaas onmogelijk om in slechts enkele paragrafen iedereen bij naam te bedanken die me de afgelopen jaren heeft bijgestaan. Daarom wil ik starten met in het algemeen iedereen te bedanken die me gedurende mijn doctoraat op de een of andere manier heeft bijgestaan. Ook al sta je misschien niet expliciet in deze tekst vermeld, wees zeker dat ik dankbaar ben voor al je hulp en steun.

De invloed van mijn promotoren is natuurlijk wel het meest zichtbaar in dit doctoraatswerk. Ik wil dus eerst en vooral mijn promotoren Bjorn en Koen bedanken om mij de kans te bieden om te doctoreren en om mij al die jaren bij te staan in mijn onderzoek. Als Koen mij destijds niet had aangenomen als student en mij op weg had geholpen, en als Bjorn de me de laatste jaren niet verder had geleid en had bijgestaan met kritische maar hulpvolle opmerkingen over mijn onderzoek en tekst, was dit werk er nooit gekomen.

Furthermore, I would also like to thank the other members of my exam committee: Christian Collberg, Lieven Eeckhout, Roberto Giacobazzi, Eric Laermans, Bart Preneel, and Luc Taerwe. I would especially like to thank Christian and Roberto for traveling all the way to Ghent for my internal defense despite their busy schedules.

Ik bedank ook graag het FWO–Vlaanderen voor hun financiering. Financiële ondersteuning is echter niet de enige ondersteuning die ik gekregen heb: er was ook goede technische ondersteuning. Daarbij wil ik zeker Marnix, Michiel, en Ronny bedanken voor alle hulp die ze mij geboden hebben.

Een doctoraat doe je (gelukkig) ook niet alleen: er zijn de vele collega's die de afgelopen jaren een pak aangenameer gemaakt hebben. Eerst en vooral is er natuurlijk het 'Security Team' waarmee ik een kantoor deel: Christophe, Jeroen, Ronald, en Stijn. Ondanks dat ze

jaren met mij op slechts enkele vierkante meters samen hebben doorgebracht, zijn ze me gelukkig nog niet hélemaal beu. We doen in het System Software Team echter niet enkel aan beveiliging. Er zijn er ook de collega's van 'de andere bureau'. Met sommigen heb ik nog een tijdje op hetzelfde kantoor gezeten: Jonas, Niels, en Panagiotis. Anderen kwamen na onze bureausplitsing: Hadi, Henri, en Tim. Allemaal bedankt voor de fijne samenwerking de afgelopen jaren.

Collegialiteit stopt echter niet meteen bij de grenzen van wie je promotor is. Samen met Francis en het hele WELEK-team heb ik veel genoeg gehaald uit het helpen organiseren van de robotcompetitie. Met Kenzo heb ik dan weer elk jaar met veel plezier de oefeningenlessen computerarchitectuur gegeven.

Een (bijna) vast onderdeel van mijn dagen waren de middagmaaltijden in de Brug. De vele middagen samen met de collega's van -3 en studenten waren een gezellig moment om elke keer weer naar uit te kijken.

De afgelopen jaren waren ook op persoonlijk vlak een grote verrijking, meer dan ik vooraf had durven voorspellen. Niet alleen heb ik er geen moeite meer mee om spontaan voor grote groepen mensen iets (proberen) uit te leggen, maar ik reis met een minder absolute tegenzin met het vliegtuig. En misschien het meest tegen alle verwachtingen in: ik ben begonnen met sporten in de vorm van aikido en zwemmen, en ik ben dat ook blijven doen. Daardoor heb ik zelfs een tijdje een appartement in Gent gedeeld met mijn appartementsgenoot Cedric. Maar veel belangrijker dan dat is dat ik daardoor ook veel nieuwe vrienden gemaakt heb: Bojan, Kate, Michiel, Michelle, Sarah, Tom, etc. Ik denk dus dat het duidelijk is dat ik tijdens mijn doctoraat op vele vlakken opgebloeid ben, waarvoor ik heel dankbaar ben.

Ik wil ook al mijn mede-informatici: Bert, Davy, Femke DB, Femke O, Jeroen F, Jeroen J, Niels, Pascal, en Stéphanie bedanken voor de gezellige informatica-avonden de afgelopen jaren. Ze brachten enerzijds de nodige afleiding, en anderzijds versterkten ze met de interfacultaire en inter-universitaire doctoraatsanecdotes de zekerheid dat het er elders gelijkaardig aan toe gaat bij een doctoraat.

Ik had helaas ook af en toe nood om eens te kunnen klagen over de dingen des levens. Gelukkig waren er mensen die zonder klagen naar mijn gezaag hebben willen luisteren. Michael, Robrecht, Tom, en Wim, bedankt daarvoor.

Ten slotte zou ik ook van harte mijn familie willen bedanken voor

alle steun en hulp die ik de afgelopen 27 jaar van hun gekregen heb. Mama, Papa, Bruno, meter Halet, en meter Statiesstraat, bedankt dat jullie altijd in mij zijn blijven geloven.

Bart Coppens
Gent, 7 juni 2013

Examencommissie

- Prof. Luc Taerwe, voorzitter
Vakgroep Bouwkundige Constructies
Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent
- Prof. Lieven Eeckhout, secretaris
Vakgroep ELIS
Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent
- Prof. Bjorn De Sutter, promotor
Vakgroep ELIS
Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent
- Prof. Koen De Bosschere, promotor
Vakgroep ELIS
Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent
- Prof. Christian Collberg
Department of Computer Science
University of Arizona
- Prof. Roberto Giacobazzi
Faculty of Mathematical, Physical and Natural Science
University of Verona
- Prof. Eric Laermans
Vakgroep INTEC
Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent
- Prof. Bart Preneel
Vakgroep ESAT
Faculteit Ingenieurswetenschappen
Katholieke Universiteit Leuven

Leescommissie

- Prof. Bjorn De Sutter
Vakgroep ELIS
Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent
- Prof. Christian Collberg
Department of Computer Science
University of Arizona
- Prof. Roberto Giacobazzi
Faculty of Mathematical, Physical and Natural Science
University of Verona
- Prof. Eric Laermans
Vakgroep INTEC
Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent
- Prof. Bart Preneel
Vakgroep ESAT
Faculteit Ingenieurswetenschappen
Katholieke Universiteit Leuven

x

Samenvatting

Software bevat vaak imperfecties en zelfs fouten. Eén type fout dat regelmatig het nieuws haalt, is het veiligheidslek. Een veiligheidslek kan een aanvaller toelaten om geheime informatie van gebruikers te achterhalen, of om controle te verkrijgen over de computer van een slachtoffer.

In dit doctoraatswerk bestuderen we hoe we gebruikers kunnen beschermen tegen veiligheidslekken in software. In het bijzonder zullen we de interactie tussen *variatie* in software en de veiligheid van software bestuderen. We onderzoeken twee soorten variatie: (1) variatie tussen verschillende programmaversies, en (2) variatie in de uitvoeringstijd van één enkel programma.

Variatie tussen programmaversies Nadat softwaremakers een veiligheidslek geïdentificeerd en weggewerkt hebben, zullen ze een gepatchte programmaversie naar hun klanten sturen. Bij zo een gepatchte versie zit vaak een korte beschrijving van het veiligheidslek, zonder echter aanvallers op weg te helpen bij het opzetten van een aanval. De aanvaller heeft echter meer informatie dan enkel deze beschrijving. Er is een verschil tussen de oorspronkelijke en de gepatchte versie van het programma, namelijk daar waar het veiligheidslek weggewerkt is. Een aanvaller heeft er dus baat bij om niet enkel de beschrijving van het lek te bestuderen samen met de gepatchte versie, maar om ook de gepatchte versie te vergelijken met de oorspronkelijke versie. Een aanvaller kan hierbij gebruik maken van zogenaamde *diffing tools*. Dit zijn tools die twee binaire programma's vergelijken en een lijst met corresponderende codefragmenten teruggeven.

Met onze eerste onderzoeksbijdrage tonen we hoe echte aanvalstools inderdaad op een efficiënte manier gebruikt kunnen worden door aanvallers om veiligheidslekken te vinden. We tonen aan dat we een

aanvaller die gebruik maakt van zulke aanvalstools automatisch kunnen modelleren. Dit leidt tot meerdere modellen die afhangen van welke tools en welke heuristieken de gemodelleerde aanvaller gebruikt. Deze modellen gebruiken we dan om automatisch te schatten hoeveel moeite een aanvaller moet doen om een veiligheidslek op te sporen aan de hand van een softwarepatch.

Vervolgens diversifiëren we de gepatchte binaire bestanden. Met deze techniek voegen we kunstmatig variatie toe tussen de twee programmaversies. Hoewel softwarediversiteit reeds eerder voorgesteld is, is het nog niet geëvalueerd op het ontdekken van veiligheidslekken met echte aanvalstools. Met onze schatting van de aanvalsinspanning tonen we aan dat softwarediversiteit ook effectief tegen zulke aanvallen kan beschermen, maar dat er een grote impact kan zijn op de uitvoeringstijd van de gediversifieerde programmaversies.

Onze tweede onderzoeksbijdrage is een verbetering ten opzichte van de huidige stand van zaken in softwarediversiteit. Bestaande raamwerken voor softwarediversiteit selecteren stochastisch de transformaties die toegepast worden op codefragmenten. Hierbij worden vaste kansen gebruikt om te beslissen welke transformaties op welke codefragmenten toegepast worden. Wij verbeteren dit door in het diversificatieproces gebruik te maken van de informatie die gegenereerd wordt door diffing tools. We beginnen met het automatisch vergelijken van de ongepatchte en de gepatchte programmaversies. Dit levert ons een lijst corresponderende codefragmenten op. Het doel van ons raamwerk is om het resultaat van de diffing tool minder nuttig te maken voor een aanvaller, en dit met zo weinig mogelijk overhead in programma-omvang en/of uitvoeringstijd te introduceren. Ons raamwerk zal daarbij enkel codefragmenten transformeren die aan elkaar gelinkt kunnen worden door de diffing tool. Bovendien gebruiken we ook informatie over hoe de diffing tool deze codefragmenten aan elkaar gekoppeld heeft om transformaties te kiezen om er op toe te passen. Hierna vergelijken we het gediversifieerde programma opnieuw met het niet-gepatchte programma. Dit levert ons opnieuw een lijst met corresponderende codefragmenten op, die we dan weer kunnen gebruiken als invoer voor ons raamwerk. We kunnen dit proces dus iteratief toepassen om uiteindelijk een gediversifieerde gepatchte programmaversie te bekomen die zó getransformeerd is dat de aanvalstool nauwelijks nog corresponderende codefragmenten vindt.

Variatie in uitvoeringstijd De uitvoeringstijd van programma's kan afhangen van geheime informatie, en dit via zowel controleafhankelijkheden als data-afhankelijkheden. Een aanvaller kan deze afhankelijkheden bestuderen, en hiervan gebruik maken om geheime informatie te stelen van de gebruikers van de programma's.

In onze derde bijdrage stellen we een compilertoolflow voor die automatisch programma's transformeert zodat ze beschermd zijn tegen aanvallers die gebruik proberen te maken van de uitvoeringstijd van het programma. We beschermen het programma door de variatie in uitvoeringstijd te verminderen. We evalueren onze compiler op recente Intel x86 processors.

Onze compiler maakt gebruik van kennis van tijdsnevenkanalen om de variatie in uitvoeringstijd te verminderen. We doen dit door gebruik te maken van *if-conversion*. Dit is een transformatie die controleafhankelijkheden omzet in data-afhankelijkheden. We tonen aan dat deze data-afhankelijkheden geen extra tijdsvariatie introduceren. Bovendien tonen we aan hoe controleafhankelijkheden verwijderd kunnen worden door enkel gebruik te maken van conditionele toewijzingen, ook al vertonen de getransformeerde functies mogelijke neveneffecten zoals het opgooien van fouten.

Verder bestuderen we ook twee bronnen van tijdsvariatie afkomstig van data-afhankelijkheden. De eerste bron van tijdsvariatie zijn de deelinstructions op de Intel Core 2 Duo processors. We onderzoeken verschillende strategieën om de mogelijke variatie door het gebruik van deze instructies weg te werken. De andere bron van tijdsvariatie die we onderzoeken is *load bypassing*. Load bypassing introduceert tijdsvariatie bij geheugentoeegangen, ook al zijn deze toegangen onafhankelijk van elkaar. Net zoals bij de deelinstructions stellen we ook hier voor hoe we deze tijdsvariatie automatisch kunnen verhelpen.

Het eindresultaat van onze compiler is dus een programma dat significant minder, of zelfs helemaal geen tijdsvariatie meer vertoont die afhankelijk is van geheime invoer. Hierdoor zal de aanvaller dus geen tijdsinformatie meer kunnen gebruiken om geheime informatie te achterhalen.

De resultaten van dit onderzoek zijn drieledig. Ten eerste modelleren we aanvallers die gebruik maken van de variatie tussen programma's om veiligheidslekken te vinden. We gebruiken deze modellering dan om aan te tonen dat softwarediversiteit zulke aanvallen kan bemoeilijken. Ten tweede tonen we aan hoe we betere resultaten kun-

nen bekomen dan bestaande diversiteitsraamwerken door op iteratieve wijze feedback te gebruiken van diffing tools. Ten slotte stellen we een compiler voor die rekening houdt met tijdsnevenkanalen. De door deze compiler gegenereerde programma's zullen geen variatie in uitvoeringstijd vertonen die afhangt van geheime informatie.

Summary

Software may contain imperfections and errors. One kind of software errors that often catches people's attention are security vulnerabilities. A security vulnerability may allow an attacker to extract private information from users, or to gain control of a victim's computer. This is obviously undesirable.

In this PhD work we study how to protect users against such vulnerabilities in software. We focus on how *variation* affects security, and how manipulating such variation can protect users against attacks. We study two kinds of variation: (1) variation between different versions of the same program, and (2) the variation in execution time of a single program.

Variation between different versions of the same program After a software vendor has identified and patched a vulnerability, the vendor distributes a patched version of the program to customers. This patched version usually comes with a short explanation of the fixed vulnerability to help system administrators decide whether or not to apply the patch. This explanation often does not contain practical hints for attackers. However, an attacker has more information than the description of the patch. Observe that the patched program version differs from the original program version: their behavior will differ for at least one input. This difference in behavior will be caused by a difference in the program's binary representation. Knowledge of this binary difference will lead the attacker to the difference in behavior, which will in turn lead him to the patched vulnerability itself. This gives an attacker who knows the difference between both versions a head start compared to an attacker who only has a vague description of the fixed vulnerability. An attacker can visualize the difference between binary programs with so-called diffing tools. These are tools that compare two

binaries and return a list of code fragments that are matched between them.

With our first contribution, we show that real-world attack tools can be used effectively by attackers to find vulnerabilities. We show that we can model the process of an attacker using different real-world attack tools. This leads to multiple models for an attacker, depending on which tools and heuristics the modeled attacker uses. We use these to automatically approximate the effort required by an attacker using such tools.

We then diversify the patched binaries. This is a technique that artificially increases the variation between two binaries. While software diversification has been proposed before, it has not been evaluated using real-world attack tools to recover information about a patched vulnerability. Using our approximation of attack effort with real attack tools, we show that software diversification indeed protects against existing attack tools, but that the overhead in execution time for such a protection can be large.

Our second contribution is an improvement over the current state of the art in software diversification. Existing diversification frameworks use a stochastic process to select the transformations that are applied to code fragments with the same probability throughout the binary. We use feedback from a diffing tool to iteratively guide the diversification process. We start by using a diffing tool to compare the unpatched binary to the patched binary. Our goal is that the attack tool finds as few matching code fragments as possible, at the lowest overhead. Thus, we only transform code fragments that are matched by the diffing tool. Code fragments that are not matched, are not transformed. Which specific transformations are applied to the code fragments depends on exactly how the attack tool matched the code fragments. We then iteratively use the attack tool to improve the output of the diversification framework: we compare the unpatched binary to the diversified framework and use this comparison as feedback for a new iteration of our framework. The end result is a binary that has been specifically transformed so that the attack tool finds as few matching code fragments as possible.

Variation in execution time The execution time of a program may depend on private information. An attacker can analyze how the execution time depends on private information, and use this knowledge

to extract it. He will do this by measuring the execution time of the program. The execution time of a program may depend on the private information through either its control dependencies, or through its data dependencies.

In our third contribution, we propose a compiler tool flow that automatically transforms programs so that they are protected against attackers that exploit timing variation to extract private information. We do this by reducing the timing variation that a program exhibits. We evaluate our compiler on recent Intel x86 processors.

Our timing side-channel aware compiler removes control dependencies in programs. We do this using if-conversion, which transforms control dependencies into data dependencies. We show that these data dependencies do not introduce execution time variation. Furthermore, we show how to remove control dependencies using only conditional move operations, even if the transformed functions possibly exhibit side-effects such as causing exceptions.

We also investigate two sources of timing variation due to data dependencies. We show that division instructions on Intel Core 2 Duo processors have a variable execution time, and discuss different strategies to remove this source of execution time variation from programs. We also investigated timing variation due to load bypassing. In that case we observe timing variation due to memory accesses, even if they are not dependent on each other. As with the division instruction, we propose a mechanism to automatically reduce this timing variation in our compiler.

Binaries generated by our timing side-channel aware compiler will thus exhibit significantly less, or, depending on the data dependencies left, no execution time variation at all. Thus, an attacker will not be able to use the timing information to extract private information.

The results of this research are threefold. Firstly, we show that we can model attackers that use diffing tools to exploit variation between program versions. We use this to show that software diversification techniques can effectively thwart such attackers. Secondly, we show how to improve on existing diversification techniques by iteratively including feedback from diffing tools in the diversification algorithm. Finally, we present a timing side-channel aware compiler that produces binaries whose execution time does not depend on private information.

Contents

| | |
|---|-----------|
| Nederlandse samenvatting | xi |
| English summary | xv |
| 1 Introduction | 1 |
| 1.1 Abstracting binaries into models | 3 |
| 1.1.1 Disassemblers | 4 |
| 1.1.2 Execution tracing | 7 |
| 1.1.3 Tools for abstracting binaries | 8 |
| 1.2 Characterization of patched binary code | 11 |
| 1.3 Software matching and Exploit Wednesday | 13 |
| 1.4 Software matching | 15 |
| 1.4.1 Binary patch generation tools | 15 |
| 1.4.2 Graph-based matching approaches | 16 |
| 1.4.3 Trace-based matching approaches | 21 |
| 1.4.4 Polymorphic malware analysis | 22 |
| 1.4.5 Attack tools for software matching | 23 |
| 1.5 Diversification as protection against software matching . | 25 |
| 1.6 Timing side channels | 27 |
| 1.6.1 Control flow | 27 |
| 1.6.2 Data flow | 29 |
| 1.7 Exploiting timing side channels | 30 |
| 1.7.1 Measuring timing variation | 30 |
| 1.7.2 Recovering private information | 32 |
| 1.8 Protecting against timing side channels | 33 |
| 1.9 Contributions | 35 |
| 2 The effectiveness of variation against patch-based attacks | 39 |
| 2.1 Introduction | 39 |
| 2.2 SCIMs and TIMs | 41 |

| | | |
|----------|--|------------|
| 2.3 | Heuristic attack model | 43 |
| 2.3.1 | A framework for attack models | 44 |
| 2.3.2 | Binary diffing tools | 46 |
| 2.3.3 | Additional prioritization heuristics | 55 |
| 2.4 | Diversification as mitigation strategy | 59 |
| 2.4.1 | Diversifying transformations in Proteus | 59 |
| 2.5 | Evaluation | 65 |
| 2.5.1 | Case studies | 65 |
| 2.5.2 | Representing the results | 69 |
| 2.5.3 | Effectiveness of attacks on undiversified binaries | 69 |
| 2.5.4 | Diversification | 75 |
| 2.6 | Conclusion | 77 |
| 3 | Iterative feedback-driven diversification | 81 |
| 3.1 | Introduction | 81 |
| 3.2 | Feedback-guided iterative diversification | 82 |
| 3.2.1 | Attack model | 82 |
| 3.2.2 | Diversifying transformations | 83 |
| 3.2.3 | Transformation Selection | 87 |
| 3.3 | Evaluation | 91 |
| 3.3.1 | Diffing results | 91 |
| 3.3.2 | Overhead | 95 |
| 3.3.3 | Representativeness | 99 |
| 3.4 | Discussion | 103 |
| 3.5 | Conclusion | 105 |
| 4 | Removing variation in execution time | 107 |
| 4.1 | Automatically removing control-dependent variation | 107 |
| 4.1.1 | Conditional execution of acyclic sequences | 108 |
| 4.1.2 | Cyclic control flow graphs | 113 |
| 4.1.3 | Function calls | 115 |
| 4.2 | Removing data-dependent variation | 117 |
| 4.2.1 | Timing variation due to early exit | 117 |
| 4.2.2 | Timing variation due to the memory subsystem | 119 |
| 4.3 | Implementation | 123 |
| 4.4 | Evaluation | 124 |
| 4.4.1 | Experiments | 124 |
| 4.4.2 | Register-based dependencies | 128 |
| 4.4.3 | Effectiveness | 130 |
| 4.4.4 | Efficiency | 132 |

| | | |
|----------|--|------------|
| 4.4.5 | Code size overhead | 134 |
| 4.5 | Comparison with existing techniques | 135 |
| 4.5.1 | Source-based solutions | 136 |
| 4.5.2 | Binary rewriting | 138 |
| 4.5.3 | Hardware instructions | 138 |
| 4.6 | Conclusion | 139 |
| 5 | Conclusions & Future work | 141 |
| 5.1 | Conclusions | 141 |
| 5.1.1 | Variation between program versions | 141 |
| 5.1.2 | Variation of execution time in a program | 143 |
| 5.2 | Future work | 144 |
| 5.2.1 | Variation between program versions | 144 |
| 5.2.2 | Variation of execution time in a program | 146 |

List of Tables

| | | |
|-----|---|-----|
| 2.1 | Different settings of the diversity system | 64 |
| 2.2 | Binary size characteristics of the case studies | 69 |
| 2.3 | The heuristics used for pruning the results of the attack tools | 70 |
| 3.1 | Glaucus default rule set | 89 |
| 3.2 | Diversification strategy for a function $f()$ | 90 |
| 4.1 | Statistical results of if-conversion and the elimination of variable-latency division instructions | 133 |

List of Figures

| | | |
|------|---|----|
| 1.1 | Screenshot of the BinDiff diffing tool. | 17 |
| 1.2 | Example of how functions in a call graph can be matched iteratively. | 18 |
| 1.3 | Example of how selectors and properties can be used to match functions. | 20 |
| 1.4 | Software chain from vendor to user | 26 |
| 2.1 | The attacker's return on attack investment | 40 |
| 2.2 | Attacker's tool flow to find SCIMs in a binary patch. . . . | 44 |
| 2.3 | Schematic overview of the attack tools operating on byte sequences. | 46 |
| 2.4 | Screenshot of the patchdiff2 diffing tool. | 50 |
| 2.5 | Screenshot of the TurboDiff diffing tool. | 51 |
| 2.6 | Screenshot of the BinaryDiffer diffing tool. | 52 |
| 2.7 | Screenshot of the BinDiff diffing tool. | 53 |
| 2.8 | The attacker will also study matched code fragments adjacent to unmatched code fragments. | 57 |
| 2.9 | Tool flow when distributing a patched program version using the Proteus diversification framework | 58 |
| 2.10 | An example of tail duplication. | 60 |
| 2.11 | Predicating a basic block by a two-way opaque predicate | 61 |
| 2.12 | Control flow flattening | 62 |
| 2.13 | Schematic representation of a branch function | 63 |
| 2.14 | Source code changes for three of the four patches | 66 |
| 2.15 | Source-code induced mutations in three of the four binary code patches | 67 |
| 2.16 | Detailed example of pruning and recall rates | 70 |
| 2.17 | Pruning and recall rates for bsdiff and xdelta | 71 |
| 2.18 | Pruning and recall rates for patchdiff2 and BinaryDiffer | 72 |
| 2.19 | Pruning and recall rates for patchdiff2 and BinDiff | 73 |

| | | |
|------|---|-----|
| 2.20 | Comparison of all tools on diversified and undiversified binaries | 79 |
| 2.21 | Size overhead of diversification | 80 |
| 2.22 | The overhead in execution time compared with attack tool performance. | 80 |
| 3.1 | Iterative tool flow of Glaucus for generating protected patches. | 83 |
| 3.2 | Branch function insertion | 86 |
| 3.3 | Detailed iterative diffing results for bzip2 | 93 |
| 3.4 | Coarse diffing results for bzip2 | 94 |
| 3.5 | Coarse diffing results for png_debian | 96 |
| 3.6 | Coarse diffing results for png_beta | 97 |
| 3.7 | Coarse diffing results for soplex | 98 |
| 3.8 | Binary code size of the transformed programs, relative to the undiversified program | 100 |
| 3.9 | Binary patch size of the transformed programs, relative to the undiversified patch | 100 |
| 3.10 | Execution times of the transformed programs, relative to the undiversified program | 101 |
| 3.11 | Relative execution times vs. percentage of correctly matched instructions. | 101 |
| 3.12 | Cross-validation of Glaucus with other diffing tools on bzip2. | 102 |
| 3.13 | Cross-validation of Glaucus with other diffing tools on soplex. | 102 |
| 4.1 | Parallel constant-time code to make the result of divisions constant time as well. | 119 |
| 4.2 | C code equivalent of unsigned division computation using only fixed-latency divisions. | 120 |
| 4.3 | Execution times of a microbenchmark loop with 4-byte load and store instructions executed for varying displacements between the accessed locations. | 121 |
| 4.4 | Microbenchmarks on which we evaluated our timing side-channel aware compiler. | 125 |
| 4.5 | Average (over pseudo-random inputs) execution slowdown after applying if-conversion and elimination of variable-latency division instructions. | 134 |
| 4.6 | Code size increase after applying if-conversion and elimination of variable latency division instructions. | 135 |

List of Abbreviations

| | |
|------|--|
| AST | Abstract Syntax Tree |
| BBL | Basic Block |
| CFG | Control Flow Graph |
| CG | Call Graph |
| DCF | Dynamic Call Graph |
| DDDG | Dynamic Data Dependence Graph |
| IR | Intermediate Representation |
| ISA | Instruction Set Architecture |
| PRNG | Pseudo-Random Number Generator |
| RSA | Rivest-Shamir-Adleman public key algorithm |
| SCIM | Source-Code Induced Mutation |
| SSA | Single-Static Assignment |
| TIM | Translation-Induced Mutation |
| VLIW | Very Long Instruction Word |

Chapter 1

Introduction

When people use software, they are typically unaware of all the risks that accompany it. Software is vulnerable to attacks. While the power of most vulnerabilities is limited to causing an attacked program to crash, some vulnerabilities will enable attackers to run their own malicious code on the victim's machine, or to exfiltrate private information. Two recent high-profile attacks are the Stuxnet worm and the DigiNotar break. Stuxnet is malware that uses multiple vulnerabilities in Microsoft software to spread itself without detection. It was allegedly used to break Iran's centrifuges [61]. DigiNotar was a certificate authority that signed certificates of identity to websites and to the Dutch government. An attacker broke into their system through a web server that had not been patched for a security vulnerability. The attacker was able to gain access to DigiNotar's signing server. This allowed him to sign his own, fraudulent certificates to impersonate authenticated Google servers [77].

There are different strategies with which an attacker can exploit software. One strategy is to study the target program, to find a vulnerability in it, and then turn this vulnerability into an exploit. This is the strategy used by the Stuxnet attack. Another kind of attacker will wait for other people to disclose a security vulnerability, will find a way to exploit this vulnerability himself, and then use it on his victims. This is the strategy the DigiNotar attack used. There are of course also attackers who merely re-use attack software made by other attackers using one of the previous strategies.

In this PhD work, we will study how some attacks on software can be mitigated. Our focus is on mitigating attacks that exploit the *varia-*

tion that is present in software. In particular, we consider two different kinds of variation and their impact on software security:

1. *Variation between program versions.* When a software vendor releases a security patch for a vulnerability, the patch discloses the location of the vulnerability in the program to the attacker. An attacker can find this location by comparing the vulnerable version of the program with the patched version. The attacker can then try to exploit said vulnerability because not all users immediately apply the patch.

We know this kind of attack exists because attackers sometimes publicly describe how they reversed and created an exploit for a vulnerability, solely based on the information in the security patch [96, 99, 141].

2. *Variation in execution time.* An attacker can try to correlate the execution time of a program with private information used in this program. This is a case of an attacker trying to find a vulnerability himself.

Exploiting timing variation in programs is quite old. One famous example from the days when multi-user systems were first being introduced, is checking the correctness of the log-in password of a user [93]. To check the entered password, the log-in software will compare the it character-wise with the stored password. When this software stops checking the password as soon as the first mismatching character is found, an attacker can measure how long the password check takes. Then he immediately knows if the first character matches, or the first two characters, etc. That way, the attacker converts an exponential search space into a linear one. Even though password checks nowadays typically are implemented using hashes, similar implementation problems still occur [95].

We will defend against both kinds of attacks by modifying the amount of variation that the attacker will observe. Our techniques will focus on generating more secure *binary code*. In order to describe exactly how we do this, it is useful to know how existing attacks and defenses against these attacks work. In the next sections, we will describe both the offensive and the defensive side in more detail.

In Section 1.1, we describe how to identify and interpret binary code. The attacker has to know where the code is located in a program,

and what this code does, before he can attack the program. This is the case for both exploiting variation between program versions, as well as for exploiting timing variation.

We then describe how to exploit variation between two binary program versions. To interpret the changes in the binary code caused by a binary patch, the attacker needs to understand how source code change affects the binary code. Section 1.2 describes different kinds of change at the binary level that can be caused by source code changes. Section 1.3 discusses how attackers can use the difference between the patched and the unpatched program. We then explain how an attacker can find the difference between two binary programs in Section 1.4. Section 1.5 then describes existing techniques that try to prevent the impact of such an attack.

This is followed by a discussion of attacks that exploit execution time variation. Section 1.6 describes how the execution time of a program can depend on its input. Section 1.7 then briefly describes different strategies the attacker can use to exploit these variations to extract private information. Finally, Section 1.8 describes existing techniques that reduce the timing variation of programs.

With these attack strategies in mind, we finish with the contributions this PhD work makes in Section 1.9.

1.1 Abstracting binaries into models

To attack a binary program, an attacker needs to understand at least part of the behavior of the program. The attacker can construct a model of the binary program to reason on. The first step towards making a model of the binary will be to locate and identify the instructions in the binary. These instructions can be further modeled by grouping them into more abstract representations. These representations can then be used in conjunction with a debugger to increase the understanding of the program interactively.

During the creation of a program binary by compiling and linking, the program is viewed at different levels of abstraction. These different representations are used to represent the structure of the program. They offer a high-level model on which different analyses and transformations can be applied [110]. The attacker then aims to reconstruct the structure of a program at a certain level of abstraction.

At the lowest level of abstraction, the binary consists only of the raw bytes that make up the code and data. Above this is the level of assembly, where sequences of adjacent bytes are instructions which have certain semantics. Sequences of instructions that are not split by any control flow in the program are called basic blocks (BBLs). These BBLs can be grouped into control flow graphs (CFGs). The nodes in a CFG are BBLs, while the edges correspond to the possible control flow between BBLs. Basic blocks are partitioned into functions, where each function has a separate CFG. The caller-callee relationships between functions can be modeled by a call graph (CG) in which the nodes model functions and the edges model function calls.

One approach to modeling a binary is to describe the control flow statically. This static representation of the binary takes into account all possible control flow, including control flow that does not happen when executing the binary. Software vendors can try to thwart static analyses with different techniques, which will we describe in Section 1.1.1. Another approach to modeling a binary is to only model the code that is executed. This dynamic representation can be used by the attacker to reconstruct and approximate the exact CFG.

In this section, we discuss several techniques and tools that can be used to detect and interpret the instructions in a binary, using either static or dynamic information. We will also discuss the different caveats associated with these techniques.

1.1.1 Disassemblers

The goal of a disassembler is to decode the bytes that make up the binary code into sequences of instructions. The only starting information an attacker can usually rely on is the start address of the program, and a list of sections in the program containing executable code. In some cases, this information is augmented with the names and the entry points of individual functions. This information is then used to create a list of disassembled instructions. An attacker can abstract this information by grouping the disassembled instructions into BBLs and CFGs. Disassembling binary code is not a trivial task: there are many problems and subtleties that can make disassemblers return incomplete and incorrect information. Incomplete information corresponds to false negatives, which means that there are instructions in the binary that are not decoded by the disassembler. Incorrect information is also referred

to as false positives, which means that the disassembly does not reflect the code that is actually executed when running the program. The goal of the attacker is to gain as much useful information from the binary as possible. As such, both incomplete and incorrect information can present difficulties to the attacker.

Static disassemblers assume that the binary code representing the instructions does not change during program execution. This assumption is invalid with self-modifying code, in which a program creates or overwrites instructions at run time. This means that the CFG changes through the course of the program execution. To model this, different models of executable code such as a State-Enhanced CFG are needed [9].

In this section, we describe several static disassembly algorithms and their handling of incorrect and incomplete information.

Linear sweep disassemblers Linear sweep disassemblers assume that all instructions are placed sequentially in a code segment. They compute the starting address of the next instruction using the length of the currently decoded instruction. Instructions are decoded until the end of the code segment is reached [59].

Linear sweep disassemblers analyze the whole code section. When all instructions have equal length and all memory fetches have to be aligned at multiples of this length, a linear sweep will correctly decode all instructions contained in the section. Thus, its results are complete in the absence of self-modifying code. However, when these assumptions do not hold, disassembly becomes more difficult. A decoded sequence of bytes can overlap *partially* with the instructions present in the binary. For example, the disassembler can start decoding in the middle of an instruction. Because the start address of the next instruction is computed using the length of the current instruction, once the disassembler decodes a sequence of bytes that does not correspond exactly to an instruction in the binary, subsequent bytes can be decoded incorrectly as well. It has been shown that sequences of incorrectly decoded bytes will quickly synchronize with the correct instructions in the case of, for example, x86 variable-length instructions [98]. Such synchronization problems can occur when disassembling regular, compiler-generated code. Some compilers will intersperse data in code sections, which results in the disassembler trying to decode data. This might result in a synchronization issue, possibly leading to an incorrect and

incomplete disassembly. This can be used as a defense mechanism to thwart attackers: regions of data can be interspersed with code to intentionally introduce synchronization problems [98].

Recursive descent disassemblers Recursive descent disassemblers are an extension of linear sweep disassemblers. They also decode instructions sequentially, but only individual BBLs at a time are decoded [59]. The disassembly starts at the program's entry point and exported functions. Once a control transfer instruction is detected, the disassembler can not only continue disassembling the instructions in the fall-through path of this instruction, if it exists, but it can also continue disassembling at the target address of the control transfer, if possible.

Recursive descent disassemblers only disassemble code that is statically reachable from the program and function entry points, while data blocks will not be disassembled. As such, this technique is more robust against compiler-inserted data blocks. However, this technique can still return incorrect disassembly when certain obfuscation techniques are used. Recursive descent disassemblers that assume that call instructions return to the instruction following the call can be thwarted by replacing direct jump instructions with a call that does *not* return to the next instruction. The target function of such a call is called a *branch function* [98]. Similarly, software developers can add conditional control transfers that are known in advance to always transfer the control. The fall-through block can then contain data to introduce synchronization problems in disassemblers [42]. Furthermore, techniques exist to transform control transfer instructions into non-control transfer instructions [56, 121].

A major disadvantage of recursive descent disassembly is that it can be hard to compute the target addresses of control transfers. For example, function pointers and other forms of indirection make static disassembly hard, if not impossible. In such cases, recursive descent disassemblers may return incomplete information.

Symbolic execution for disassemblers Improvements can be made to the recursive descent algorithm by increasing the resilience against certain obfuscation transformations. Kruegel et al. [91] describe how a disassembler can try to detect branch functions. Assuming that the implementation of branch functions is simple and free of side effects,

they symbolically evaluate calls to branch functions, in order to recover the original control flow with direct jump instructions.

Trying all possible start addresses Another possibility is to start disassembling at *all* possible start addresses. Using the aforementioned synchronization property of the x86 ISA, a disassembler can re-use disassembly of previously decoded memory locations once synchronization occurs. All code that is part of the program is correctly disassembled: there is no problem of computing control transfer targets, or disassembled data overlapping with instructions. Such techniques can be used in the context of binary program rewriting. They ensure that all possible basic blocks that can be executed are considered when rewriting a program [154]. This technique thus produces a superset of all instructions in the binary. While this is acceptable for program rewriting, this also presents additional basic blocks to the attacker that are *not* actually present in the binary program, which makes this technique less useful for attackers.

1.1.2 Execution tracing

An attacker can use the aforementioned methods to try to disassemble all bytes in an entire code section or program. After disassembly, he can try to reconstruct the static CFG of the program. To overcome the challenge of determining which bytes in a program represent instructions and are thus part of the CFG, attackers can also consider a dynamic CFG as an approximation of the static CFG.

A dynamic CFG can be obtained by tracing the execution of the program. Multiple dynamic CFGs can exist: they depend on the inputs to the program *during tracing*. By tracing the execution of the program, an attacker knows the starting address of each executed instruction, and can thus easily decode it correctly. A dynamic CFG can be constructed by observing which control flow transfers are taken. Furthermore, by tracing a program, an attacker can also deal with code that is generated dynamically, including self-modifying code [103, 107].

The execution of a program can be traced by instrumenting the program code [104]. Thus, the original program code executes, in addition to inserted code that reports on various properties of the runtime behavior of the program. These properties include data values that are produced by instructions, which data memory addresses are

accessed by which instructions and whether or not these memory addresses have been initialized correctly, etc. [100, 113, 128].

Another commonly used technique for tracing the program execution is to use the processor's native debugging infrastructure [82]. An attacker can interactively break the execution of a program depending on dynamic conditions. For example, he can stop the execution when certain instruction addresses are executed, when certain data is accessed or modified, etc.

Finally, the execution of a program can also be observed by simulating the whole system on which the program runs [21].

1.1.3 Tools for abstracting binaries

In this section, we give a quick overview of some of the tools that are used by both attackers and defenders to abstract binaries into a higher-level intermediate representation.

Disassemblers

gdb and objdump `gdb`¹ and `objdump`² are free software utilities most commonly used to debug and analyze Linux ELF binaries. Both can be used as a linear sweep disassembler. Furthermore, an attacker can use `gdb` as a debugger, which can act as a tracing tool. The disassembler can also be used interactively by an attacker. Since breakpoints and watch points can be set to break the execution of a program under certain conditions, an attacker can start disassembling code at program points that are known to be executed.

IDA Pro IDA Pro³ is a commercial, graphical disassembler and debugger framework. It is able to disassemble code compiled for a variety of processor architectures. The attacker can use the disassembler interactively: next to using the automatic recursive descent disassembler functionality, attackers can manually select additional starting points to disassemble, can select which parts of the program are data or code, etc. The attacker can also improve on the heuristics IDA Pro uses to

¹<https://www.gnu.org/software/gdb/>

²<http://sourceware.org/binutils/docs/binutils/objdump.html>

³<http://www.hex-rays.com/products/ida/index.shtml>

detect and disassemble functions [129, 141]. Furthermore, IDA Pro offers a graphical overview of the CFGs and CGs that have been reconstructed. It offers a range of productivity features that help an attacker in exploring and understanding the code. Another important feature is its extensibility and the available plug-ins. It is scriptable through a Python interface, C++ bindings for native modules, and a custom scripting language, all of which are used by different third-party plug-ins and scripts.

Immunity Debugger Immunity Debugger⁴ is a debugger targeted towards the security industry for writing exploits and analyzing malware. It provides easy interfacing with fuzzers and other exploit development tools. For example, it is possible to use the Immunity debugger to fuzz kernel drivers [134]. It has special support for analyzing the program's heap memory. A Python API allows users to extend and automate its functionality.

OllyDbg OllyDbg⁵ is a binary code analysis tool for Windows, featuring the ability to disassemble, debug, and understand a program. It is commonly used to reverse engineer binaries. It uses information from Windows API calls to infer variable types on the program's stack. OllyDbg can be extended through a binary plug-in interface. Currently, it only supports 32 bit Windows binaries.

vdb vdb⁶ is a debugger written in Python using the vtrace API. It is targeted towards the security industry, and offers users both a GUI and a Python API for scripting. It supports many target platforms for disassembling binaries and debugging programs, such as Windows, Linux and Android. An interesting addition to using vdb is *vivisect*⁷. This is a fully programmable binary analysis framework in Python.

WinDbg WinDbg⁸ is a debugger with a linear sweep disassembler that is part of Microsoft's Debugging Tools for Windows. It offers de-

⁴<https://www.immunityinc.com/products-immdbg.shtml>

⁵<http://www.ollydbg.de/>

⁶<https://code.google.com/p/vivisect/>

⁷<https://code.google.com/p/vivisect/>

⁸<http://msdn.microsoft.com/en-us/windows/hardware/gg463009.aspx>

velopers the ability to debug processes remotely over a network connection, and to debug kernel-mode driver code. It supports debugging native 32-bit and 64-bit Windows binaries as well as managed .NET binaries.

Execution tracing

Anubis Anubis⁹ is a dynamic analysis framework for malware samples. In addition to execution tracing, it emulates a whole system to observe the interaction between the malware and the host system.

Diota Diota¹⁰ is a dynamic instrumentation framework that explicitly handles self-modifying code [104]. Diota is also used for vertical instrumentation, in which Diota instruments a Java Virtual Machine together with the Java programs the virtual machine executes [102].

DynamoRIO DynamoRIO¹¹ is another dynamic instrumentation framework. Its ability to dynamically detect out-of-bounds memory accesses has been used in commercial security products. It has also been used to instrument kernel code [62].

Dyninst Dyninst¹² is a dynamic instrumentation framework that can handle obfuscated binary code. It has also been used to apply security patches to programs at run-time [137].

FIT FIT¹³ is a static link-time instrumentation toolkit [50]. Thus, it allows users to rewrite programs without introducing a run-time overhead to analyze and modify the executed code. However, because it is a static tool, it does not support instrumenting self-modifying code.

Pin Pin¹⁴ is a dynamic instrumentation framework by Intel [100]. It offers a higher-level programming API than the other toolkits. It is used

⁹<http://anubis.isecclab.org>

¹⁰<http://www.elis.ugent.be/diota>

¹¹<http://www.dynamorio.org/>

¹²<http://www.dyninst.org/>

¹³<http://www.elis.ugent.be/fit/>

¹⁴<http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>

for the creation of program profiles for computer architecture research.

Valgrind Valgrind¹⁵ [113] is a dynamic instrumentation toolkit. It is mainly used by application developers debugging memory issues, such as identifying invalid memory accesses and finding large allocation sites.

1.2 Characterization of patched binary code

When a program distributed in binary form needs to be patched, the developer patches the source code, recompiles that code, and distributes the resulting binary code either as a complete new binary or as a differential update, i.e., a binary code patch. One of the attacks we study in this PhD work is a patch-based collusion attack. In such an attack, an attacker will study the update for a security vulnerability, and use this to engineer a security exploit itself. Since the attacker can generate the patched binary from the patch and the original program, we will assume that the attacker has the patched binary as well as the original binary.

It is the goal of the attacker to find the binary code that corresponds directly to the changes in source code. We call such changes in the binary code the *source-code induced mutations* (SCIMs). We give a formal definition of such mutations in Chapter 2.

Mutations to binary code come in the form of inserted, modified, and deleted instructions. New instructions can be inserted directly as the result of a source code patch, e.g., when new functionality is added to a program or when additional validity checks on input data are added. Instructions can get modified directly as the result of a source code patch as well. For example, a branch-if-zero conditional branch opcode can be modified to become a branch-if-not-zero opcode, or one of the operands of an instruction can be modified as part of an off-by-one security fix. Instructions can also be deleted as a direct consequence of source patches, for example when obsolete functionality is replaced. These direct insertions, modifications and deletions correspond to source-code induced mutations.

Source code patches can also introduce insertions, modifications and deletions of instructions indirectly, i.e., as a result of the com-

¹⁵<http://valgrind.org/>

piler's application of global code analyses, global code optimizations, and global code generation techniques. Those mutations are typically spread throughout the binary code. Some important cases are:

- When code is inserted or deleted, other code is moved around in memory. As a result, references to moved code, under the form of both absolute and relative addresses change. These addresses include the target addresses encoded in function calls and branch instructions.
- In order to exploit micro-architectural optimizations such as caches, a compiler will try to align code in memory to cache line boundaries. This process is called padding, which consists of adding the appropriate number of filler bytes, such as no-op instructions in between code fragments. When code moves because of insertions or deletions, the amount of padding will also change.
- Changes to a function's size may make the compiler switch between not inlining, full inlining, partial inlining, and cloning a function at some of its call sites.
- When profile-guided code optimizations are applied, the profile of the structure of the code may change, which can result in code being reordered and conditional branches being inverted. For example, a branch-if-less-than can become a branch-if-greater-or-equal.
- The addition of new variables in a source code fragment may increase register pressure, causing unrelated variables to be spilled on the stack.

These kinds of minor and major mutations do not affect the semantics of the application, so they correspond to purely syntactic mutations that are not relevant to an attacker. We call these mutations *translation-induced mutations* (TIMs). To focus their effort on the relevant code, attackers will rely on binary patching and matching tools. Their goal is to automatically filter out as many instances of the translation-induced mutations as possible, while not overlooking any relevant SCIMs.

This is not a trivial task. It is obvious that the attacker will have to analyze more than the mutated instructions to determine whether a mutation is source-code induced or translation induced. For example,

in some cases the attacker will have to decide whether potentially inlined code is semantically equivalent to the original code. This is complicated by the fact that an inlined function is typically optimized for its unique, known calling context, whereas the original outlined code will be more generic.

Even when only encoded addresses in instructions change in some code fragment, this can have different causes. On the one hand, the change can be a syntactic one, for example due to code motion. On the other hand, the instruction can point to a different address because the corresponding source code has changed. For example, when a call to a function is changed in the source code, the encoded address of that call will have changed as well, in which case the change is a semantic, source-code induced one, rather than a translation-induced one. To distinguish between these cases, the attacker might have to perform a more global analysis.

By separating the syntactic changes from the semantic changes, the attacker can try to deduce the vulnerability in the unpatched program.

1.3 Software matching and Exploit Wednesday

Attackers can study the changes between program versions to find and exploit the patched vulnerabilities. A particularly popular target of such attacks are Microsoft's patches for Windows. Every second Tuesday of the month, *Patch Tuesday*, Microsoft releases binary software updates. These updates include security patches, most of which are documented to inform system administrators what they are vulnerable to. Microsoft typically words this without giving concrete hints to hackers as to how to exploit these vulnerabilities. Their descriptions do not always match the vulnerability being patched, however, and sometimes the patched vulnerabilities are not mentioned at all [46].

As soon as attackers get their hands on the binary code patches, they start inspecting them in preparation of *Exploit Wednesday*. This term refers to their window of opportunity to target the users that did not yet apply the patch. In some cases, binary code inspection also allows attackers to identify incorrectly patched security issues. This then allows attackers to target users who did apply the patch, but were left vulnerable because the fix was not complete. There has been at least one case where the analysis of a security patch showed that it did not cover all possible inputs on which a vulnerability could be triggered [60].

Brumley et al. [32] demonstrated that in some cases exploits can be devised without any manual analysis or human understanding of the patched code. All their attack needs is an accurate identification of the modified binary code fragments before and after the patch. On that basis, it suffices to apply a specific form of constraint solving techniques called fuzzing on the program inputs to generate attacks fully automatically. This and all other published (automatic or manual) patch-based attacks that we are aware of assume that it is easy if not trivial to identify the relatively few relevant differences between the unpatched and patched versions of a binary. Many authors simply do not even consider it worthwhile to discuss how they identify the patched code fragments. Others discuss it very briefly. Most describe the use of diffing tools that point the attacker to mutated instructions. For example, Protas and Manzuik [123] briefly describe their use of IDA Pro and BinDiff diffing tool to analyze undiversified Microsoft patches for Windows. Brumley et al. mention their use of the e-Eye Binary Diffing Suite to analyze the syntactic difference between the two binaries. Oh [116] observes that the engineering of most security exploits starts with the manual or automatic analysis of the differences created by security patches, and briefly describes how EBDS and DarunGrim can be used to analyze those differences. Many other security researchers, hackers, and hobbyists tell a similar story [70, 76, 83, 96, 99, 109, 139, 141, 145, 148], conveying that spotting the relevant differences is trivial with the existing tools.

Finally, we would like to note that while we will only study binary security patches in this PhD thesis, similar attacks are a problem in the open source world. Attackers can study commits made to public repositories to find out which of these commits fixes a security issue, and then make an exploit for those vulnerabilities. They can also try to correlate commits with information from the software project's bug trackers. They can then focus only on those commits that contain security fixes. Attackers are thus able to create exploits to known vulnerabilities before users have a chance to download patched binary programs [19].

The described attacks show that attackers can indeed extract enough information from the difference between unpatched and patched binary programs to create exploits.

1.4 Software matching

Attackers want to identify the SCIMs as fast as possible because their window of opportunity closes quickly. Thus, they will use tools that automatically prune the instructions that are unchanged between program versions. Such tools are called matching tools or diffing tools. Some of these tools have been designed for other applications, but can nonetheless be used by an attacker. In this section, we discuss related work relevant to the matching of different versions of binary programs.

1.4.1 Binary patch generation tools

A first type of matching tool are binary patch generation tools. These tools are primarily used to reduce bandwidth costs when distributing patched versions of software. These tools are tailored towards the problem of generating small binary patches. The binary patches encode the differences between the two versions as efficiently as possible. Attackers could try to use the encoded differences to point them to the source-code induced mutations. The usefulness of these tools both as a tool to create small binary patches, and as a tool to pinpoint SCIMs, depends on how they encode translation-induced mutations.

The open source tool `xdelta`¹⁶ is a binary patch tool that encodes patches in the the VCDIFF format as specified in the RFC 3284 [90] standard. According to this standard, a patched binary can be reconstructed from the unpatched binary using only two kinds of operations: copying fragments from the original binary, and adding raw data. This addition of raw data can be compressed using a form of run-length encoding. Thus, a property of the VCDIFF format is that, e.g., a code fragment with only a changed offset needs to be split up into different copy-and-add operations. This problem is addressed in `bsdiff`¹⁷, which copies code fragments from the original binary that are similar to the patched binary [118]. The remaining differences are then patched separately after a decompression using the `bzip2` compression algorithm¹⁸.

The potential similarity between two binaries is even further exploited by Google's Courgette binary patch system [122]. While Courgette is designed to patch multiple executable and/or data files at once,

¹⁶<http://xdelta.org/>

¹⁷<http://www.daemonology.net/bsdiff/>

¹⁸<http://www.bzip.org/>

its innovation lies in additional processing for binary code patches. Binary code is disassembled, and it is the disassembled plaintext for which the patch is generated. Because the disassembly is a more abstract representation of the code, syntactic changes such as changed offsets are not included in the resulting patch.

1.4.2 Graph-based matching approaches

The algorithms used in binary patch tools operate at a low level of abstraction. At a higher level of abstraction there are tools that use graph-based binary matching algorithms. Their purpose is not to generate small patches between programs, but rather to identify at a higher level which code fragments are related between program versions. Such tools typically come with some kind of graphical user interface that allows users to visualize the relations between the code fragments of both program versions. Figure 1.1 is a screenshot of the BinDiff diffing tool. An attacker has asked BinDiff to show the matching instructions in a single pair of matched functions. As can be seen in the figure, the attacker can easily see which code fragments are matched between the program versions and which code fragments have been added.

The higher-level algorithms compare high-level structures such as control flow graphs, as opposed to assembly or code bytes. This often offers a more structured view of the relevant differences between the two program versions. Because of their graph-based nature, they are less prone to changes in the program layout.

Graph-based algorithms try to find graph isomorphisms between subgraphs of the control flow graphs and call graphs of programs. Since the programs are actually different, no full isomorphism will be returned. Nodes for which no mapping can be found are considered to have changed. Furthermore, as opposed to regular isomorphism construction algorithms, these algorithms are constrained to matching nodes with the same properties. This is to prevent that subgraphs with similar structure but different behavior will be matched.

Sabin [131] constructs the isomorphisms for Windows binaries by first enumerating the entry points of functions using the function export table. These functions are matched by name. Instructions in matched functions are further matched by iteratively comparing pairs of instructions during a traversal of the functions' CFGs. These instructions are compared and assigned a match strength. Different

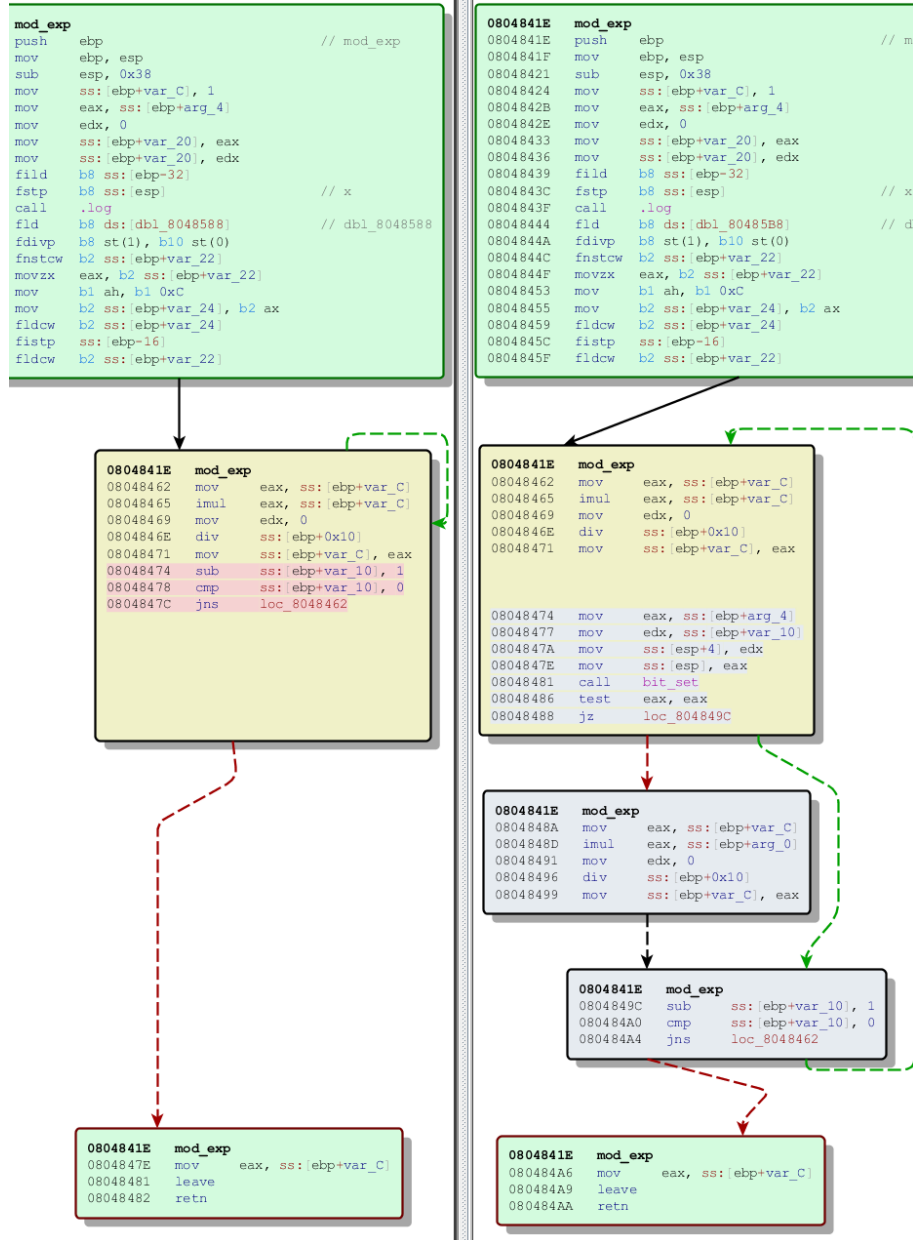


Figure 1.1: Screenshot detail of BinDiff's user interface showing the CFG of a matched function. BinDiff identifies BBLs that match with green boxes, partially matched BBLs with yellow boxes. Unmatched instructions have a grey background.

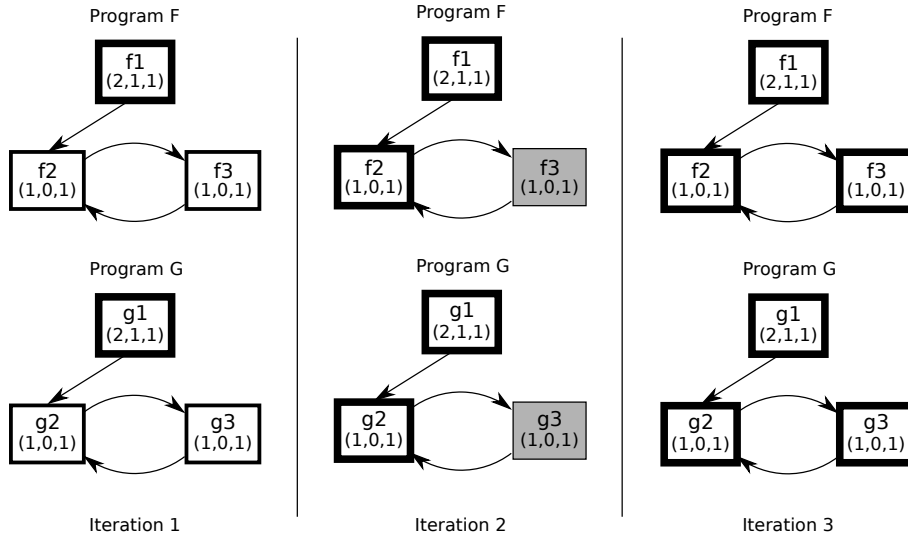


Figure 1.2: Example of how functions in a call graph can be matched iteratively based on function signatures. Each box represents a particular function. A bold border denotes a function that is matched, a grey box denotes a function that is skipped when comparing signatures.

match strengths are returned. The highest match strength is the full match; it is assigned to pairs of instructions that are (1) completely identical, or (2) have identical mnemonics. When the mnemonics are identical, either the instructions must not have any operands, or their encoded-but-different integer operands lie in the program's address space. Other instruction pairs are considered near matches when they have the same mnemonic and the same types of operands (for example, registers or immediates). Finally, no-operation instructions and direct jumps with a single successor are ignored.

BMAT [151] is a binary matching tool that has been developed primarily to reuse profile information in subsequent builds. Like Sabin, the authors of BMAT initially match functions using their names. Basic blocks are matched as well. For BBLs, initially different kinds of hashes are computed on the instructions of the BBLs, and BBLs with the same hash are matched. Finally, the CFGs of the functions are traversed, where BBLs that are in equivalent positions in the control flow are matched.

Flake [64] constructs the isomorphism by matching the programs structurally. The CGs and the CFGs are compared to match the program versions, ignoring individual instructions. He employs iterative

refinement of mappings. Initially, all functions are assigned a 3-tuple containing the number of BBLs, the number of edges in their CFGs, and the number of edges in the call graph originating from that function. Functions are partitioned by their 3-tuple. If a group contains exactly one function of each program version, these functions are considered a match. The set of matched functions is then used as a base for iteratively improving the matched function set. In subsequent iterations, functions are only considered when they have call edges originating from already matched functions. This is illustrated for the call graphs of Figure 1.2. For example, programs F and G each have one function with the signature $(2,1,1)$. This means that they both have 2 BBLs, with one edge between those BBLs, and one outgoing edge in the CG. Since they are unique in both programs, they are immediately matched (indicated by the bold box around the function). The algorithm cannot distinguish between the other functions because their signatures are identical. In the next iteration, only the functions with an edge from the already matched functions are considered. There is only a single possible match for f_2 , being g_2 . In the third iteration, we again consider the unmatched functions that have call edges originating in the already matched functions. Now there is only a single possible match for the function f_3 , being g_3 , after which all functions have been matched.

Dullien et al. [57] extend and generalize the work of Flake in two significant ways. Firstly, their algorithm generalizes the partitioning of functions with 3-tuples by considering generic *selectors* and *properties* for matching graphs. A selector maps a single node from graph A to the unique most similar node of graph B, if it exists. An example of such a selector is using the 3-tuple from Flake to find a unique match in the second binary. Properties are functions that, when given a graph, return a subset of that graph's nodes. They can thus be used to reduce the size of the graphs fed to the selectors as input. For example, a property for CGs could return only the nodes, i.e. functions, of the input CG that contain a recursive function call. Another possible property is the use of exported function names. The goal of reducing the graph is to reduce the number of identical 3-tuples for a selector, improving the chances of finding a unique match. This is illustrated in Figure 1.3. In this example, the attacker compares two programs F and G with three functions each. The selector is the number of edges per function. Since f_3 and g_2 are the only functions in each program that have exactly one edge, these two functions must match. However, we cannot match f_1 or f_2 because there are two distinct functions with zero edges in G . How-

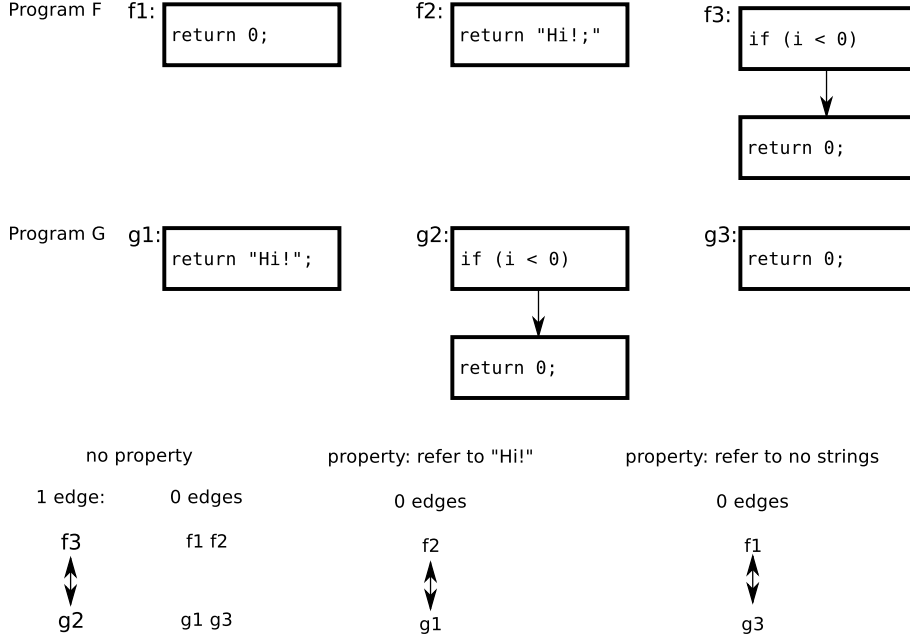


Figure 1.3: Example of how selectors and properties can be used to match functions.

ever, we can use the property *‘this function refers to the string “Hi!”’* to limit the set of functions. This property restricts the set of functions to $f2$ and $g1$. Thus, $f2$ has exactly one corresponding function with zero edges in G : $g1$. Similarly, considering functions that *‘do not refer to any string’*, the only match with zero edges for $f1$ is $g3$, after which all functions are matched.

The second extension Dulien et al. add to the work of Flake, is that they do not only consider matching functions, but also apply the same technique to the BBLs contained in the matched CFGs, as well as the instructions in these BBLs. Initially, functions are matched using function-specific selectors and properties. Then, the basic blocks of matched function pairs are again matched with selectors and properties designed for matching basic blocks. Finally, instructions of matched basic blocks are themselves matched. This work was extended by the authors of BinSlayer [105] to include other graph matching algorithms.

BinHunt [72] uses symbolic execution and theorem proving in addition to comparing the control flow graphs and call graphs of programs. BinHunt does a pairwise comparison of all functions from both programs. When comparing two functions, the maximum common sub-

graph of their CFGs is computed. The BBLs in the common subgraph correspond to matched BBLs. As a second match step, these matches are verified by proving or disproving the semantic equivalence of the matched BBLs. A theorem prover is used to prove that for all inputs, the matched BBLs compute the same output. Because the registers used for input and output may have changed between program versions, all possible combinations of input and output registers are tried. This can be computationally very expensive. As a solution to this, BinJuice was proposed as a faster way to find similar BBLs by comparing their semantics [92]. First, the semantics of BBLs is extracted. Next, the semantics is generalized into ‘juice’, which forms a semantic template of the code. The juice of different BBLs is then transformed further to efficiently find BBLs with similar semantics.

1.4.3 Trace-based matching approaches

Trace-based matching approaches collect information about the execution of a program, such as dynamic control flow, values produced, addresses referenced and data dependencies exercised.

Techniques based on compact representations of dynamic program slices [156] have been evaluated by comparing unoptimized and optimized versions of a program [155].

Similar techniques have been used to compare original and obfuscated versions as well. Nagarajan et al. [111] describe a technique that consists of two steps: an interprocedural matching step and an instruction matching step. The goal of the first step is to produce a mapping between the functions of two program versions. To enable this matching, each function is associated with a signature. By comparing signatures, *compatible* functions can be determined. The compatible functions are then matched using the structure of *dynamic call graphs* (DCGs) of the two executions, which essentially yields the function mapping. In the second step, an attempt is made to match the instructions within the matching functions. To enable this matching, each instruction is associated with a dynamic signature based on the values it produces. By comparing signatures, compatible instructions can be determined. The compatible instructions are then matched using the structure of the *dynamic data dependence graphs* (DDDGs) of the two program versions. The DDDGs are matched using the iterative algorithm discussed by Zhang et al [155]. First, the root nodes of the DDDGs are matched

by comparing the signatures. Then, the interior nodes of the DDDGs are matched using an algorithm that iteratively applies two passes. In the forward pass, nodes, all of whose parent nodes match, are in turn matched. In the backward pass, nodes that have at least one child that is matched, are in turn matched. Repeated iteration of each of these passes iteratively refines the instruction matches.

Another technique to match different obfuscated versions of programs was developed by Anckaert [7]. As opposed to Nagarajan et al., Anckaert does not try to detect and map functions, but matches only basic blocks and instructions of both programs. This is done using a matching framework that iteratively combines the different results of fuzzy classifiers. At first, classifiers are used that match BBLs of both programs. The matches of the BBLs are then propagated to the instructions contained in them, after which further classifiers are iteratively applied to the instructions. The exact steps and classifiers applied can vary, but the proposed evaluation of this framework combines information from instruction syntax, produced data values, execution count and order, system call information, and local control flow and data flow dependencies. Let us give a small example of how this framework can iteratively combine this information. First, the existing classifiers can match all BBLs that contain the same system call. Next, matches are added for all BBLs that have distance 1 from matched BBLs in the dynamic CFG. Finally, all pairs of matched BBL that do not have the same execution count are removed from the set of matches. Such classifiers can be further extended and combined into matchers that produce matches for individual instructions.

1.4.4 Polymorphic malware analysis

One specific context in which software matching techniques are playing an increasingly important role is the analysis of polymorphic malware [10]. Such malware tries to avoid being detected by mutating itself, thus thwarting tools that rely on static properties or code signatures. Anti-malware tools try to abstract these mutations, which can lead to faster or better classification of malware samples [20].

To improve the quality of the matching techniques when applied to polymorphic malware, code normalizers can be used [33, 38, 147]. These tools try to restructure the code into a normal form. Different polymorphic forms of the same code are normalized to the same nor-

mal form, which should improve the quality of the matching.

The polymorphic nature of this kind of malware typically manifests itself in the code, but not in its data structures. This has led to the emergence of techniques that match programs based on their data structures and shape rather than on their binary code [48].

1.4.5 Attack tools for software matching

Some of the diffing algorithms we described above have found their way into tools that are commonly used by attackers. As already explained, attackers will use these tools to prune matched code, so they can focus on code that is likely to have changed. These tools are also called *diffing tools*.

All of the tools mentioned here are plug-ins for the IDA Pro disassembler. As mentioned earlier, this is a recursive descent disassembler, which can return both incorrect and incomplete disassembly information. Furthermore, it may construct incomplete control flow graphs in which basic blocks are mistakenly disconnected from the others and in which many blocks mistakenly form additional entry points of different functions. When these types of mistakes are made, the diffing plug-ins obviously face a difficult if not impossible task.

BinDiff BinDiff¹⁹ is a commercial plug-in for IDA Pro based on the work of Flake and of Dullien et al [57]. It allows the attacker to find the differences at the instruction level. A set of predefined properties and selectors for matching functions and basic blocks is offered, which the user can choose from. The authors of BinDiff defined a default list of properties and selectors [159]. This default list contains different properties related to the call graph and control flow graph, but also call sequences, position-independent hashing of opcodes per basic block and instruction counts. The correctness of the matches returned by the different properties varies significantly. A rough guide to their reliability is offered by the authors of BinDiff. A final confidence and similarity score is assigned to the matched functions and displayed to the user. However, the BinDiff authors acknowledge that such scores are only a very rough indication and should not be relied on too heavily.

¹⁹<http://www.zynamics.com/bindiff.html>

EBDS, Darungrim, and BinaryDiffer The eEye Binary Diffing Suite²⁰ (EBDS) is an open source suite of tools that can be used for matching binaries. Its main use is to perform a batch comparison between two versions of an entire Windows installation. By computing hashes on all files, it finds the files that have been modified. For binary files, a list of added functions in the changed files is generated by comparing the lists of exported functions contained in the changed files. These functions can then be compared at the basic block level by using the Darungrim part of the suite, which is also available as a Python-based plug-in for IDA Pro that can be used without using EBDS²¹. There also exists a C++ re-implementation of this IDA Pro plug-in called BinaryDiffer²². It uses a combination of fingerprinting and matching on control flow graphs, data flow, and call graphs to produce its diffing results.

TurboDiff TurboDiff²³ is a free plug-in for IDA Pro. It matches functions and basic blocks using information from the control flow graph, the call graph, references to strings and by computing hashes on basic blocks. It considers functions identical if the functions have the same CFG, have the same checksum and number of instructions for each BBL. Functions are *suspicious* when the CFG is identical, but if hashes or the number of instructions differ in at least one BBL. Otherwise, functions can be considered *changed* when they can be matched using CG or string references. Individual BBLs can be (partially) matched based on their checksums and number of instructions.

patchdiff2 patchdiff2²⁴ is a free plug-in for IDA Pro. It only matches whole functions, not basic blocks or instructions. It does this by generating and comparing function signatures based on the instructions contained in the function, in string references, and in function references.

While these diffing tools all have different user interfaces and different underlying algorithms, at the most basic level their user experience is the same. A user selects two binaries to be matched; the tool then

²⁰<http://www.eeye.com/resources/security-center/research/tools/eeye-binary-diffing-suite-ebds>

²¹<http://www.darungrim.org/>

²²<https://code.google.com/p/binarydiffer/>

²³<http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=tool&name=turbodiff>

²⁴<https://code.google.com/p/patchdiff2/>

presents a list of functions that match and a list of functions that do not match. These functions can then be further inspected by viewing their CFG either in the tool itself, or in IDA Pro. However, it is unknown to the attacker whether or not the matched functions have been correctly identified.

1.5 Diversification as protection against software matching

Solutions have been proposed to thwart diffing and matching tools. For example, it has been observed before that applying obfuscation transformations to a binary makes it harder for a diffing tool to produce useful results [32, 116].

In this PhD work, we will use a technique called software diversity to defend against attackers using diffing tools. With software diversity, various transformations (including but not limited to obfuscation transformations) are applied to a binary to add artificial syntactic differences without changing the program's semantics.

Software diversity (or individualization) was first proposed by Cohen under the term *program evolution* to defend against malicious code attacks [40]. When performing a malicious code attack, the attacker has found a vulnerability, and will try to exploit this vulnerability. He does this by sending specially crafted input to the target program that causes it to execute his own code. This requires the attacker to have knowledge of how the target program will behave on the victim's computer. Software diversity makes this harder for an attacker by randomizing certain aspects of the program's internals, making it harder for an attacker to model the target system and to perform the attack. Numerous transformation techniques have been presented, including control flow transformations [7], memory layout randomization [24, 66] and instruction set randomizing [18, 84]. Other research assumes the presence of diversity and studies the assignment of distinct software packages to individual systems in a network [114] or uses different versions in a framework for detection and disruption of attacks [47, 132] similar to N-version programming for fault tolerance [16].

Software diversity can be applied at different points in the life-time of a program. This is illustrated in Figure 1.4. The first point at which software diversity can be introduced is at the source code level. This

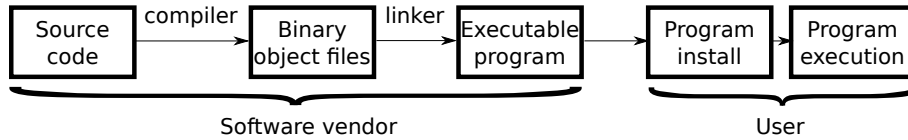


Figure 1.4: Different positions in the chain from software vendor to an end-user at which software diversity can be applied.

can be done by applying a source-to-source transformation on the program's source code before passing it through the compiler [97]. This has the disadvantage that the compiler can undo some of the transformations applied to the source code before generating the binary code. The compiler itself can also be used to introduce diversity to the program [67, 130]. After the binary code has been generated, diversity can also be added through binary rewriting, such as link-time rewriting [7, 97]. When the application is installed by the user, diversity can be introduced as well by rewriting the program at install time [112]. Finally, further changes can be introduced at run time. For example, the program's memory layout can be randomized by loading code and data at different memory addresses each time the program is executed [24]. Of course, all of these techniques can also be used together.

Software diversity has also been used as a protection mechanism against a malicious host. In that case, some sensitive software is run on a host computer to which an attacker has full access. Existing work focuses on randomization before distribution. Anckaert et al. propose to rewrite the program in a custom instruction set and to ship it with a matching virtual machine [8]. This approach turns out to have a significant performance overhead. Rather than rewrite the entire program into a virtual machine instruction set, De Sutter et al. replace infrequently occurring x86 instructions with instruction sequences consisting of more commonly used instructions [52]. In essence, they rewrite the code to a custom instruction set that is a subset of the instruction set that the processor supports natively. This reduces the performance overhead. A different approach to rewriting code fragments is to rewrite the data, rather than solely the instructions. Zhou et al. transform code to use different, but equivalent operations on 32-bit integers [158]. More application-specific diversity mechanisms to change data have been proposed as well. Using white-box cryptography, program developers can create randomized implementations of encryption algorithms that are harder to extract the private key from [36, 37].

Software diversity has been proposed as a solution against patch-based attacks. However, their effectiveness against attackers using real attack tools has not yet been investigated.

1.6 Timing side channels

So far, we discussed how attackers can exploit variation between program versions. However, we also investigated mitigations against attackers using the variation in execution time of programs. In such attacks, an attacker is only interested in differences that can be observed when executing the same program multiple times. These variations can be measured and used to infer private information that an attacker should not have been able to access. There are various sources of such intra-program variation. In cryptography, such variation is also called a side channel.

The variations that can be measured by an attacker include, but are not limited to, execution time [86], temperature [28], power consumption [87], electromagnetic radiation [5], etc. In this PhD work, we will only consider variation in execution time of cryptographic code.

This section discusses various sources of variation in program execution time. Only the variation in execution time that depends on private information is considered. An example of such private information is a private key in cryptography.

1.6.1 Control flow

The exact path that a program's execution follows can depend on private information. Consider, for example, the following naive implementation of the left-to-right square-and-multiply algorithm for modular exponentiation, which can be used in implementations of public-key cryptographic systems [71]:

```
1  result = 1;
2  i = log2(exponent);
3  do {
4      result = (result*result) % n;
5      if (bit_set(exponent, i))
6          result = (result*a) % n;
7      i--;
8  } while (i >= 0);
```

This do-while loop iterates over the bits in the exponent, and for each bit the `result` variable is updated depending on the value of that bit.

We observe that the control flow depends on the `exponent` variable on line 4. Therefore, the execution time on most modern processors depends on the number of ones in the binary representation of the exponent. In the case of decrypting a ciphertext using the RSA algorithm [126], this information is private. Thus, the execution time will depend on this private information, possibly leaking this information. Furthermore, if the execution time of the multiplication and division operations also depends on their arguments, even more private information can leak [86]. We now describe different ways in which the program's execution time depends on control flow.

Instruction count and mix The most obvious source of timing variation stems from a difference in the count and mix of executed instructions. For example, in the code example above, the control flow depends on the private key.

Hidden state Most modern processors have an internal state that does not affect the operational semantics of the instructions being executed, but can introduce timing variation nonetheless. In fact, most of them have been introduced to speed up the execution of programs, and will thus have an influence on the execution time of programs by design [80, 136]. These include, but are not limited to:

1. *Instruction cache.* Processors have an instruction cache that speeds up accesses to recently fetched instructions. Depending on whether or not an instruction is cached, and in which of the caches it resides, the latency to start executing this instruction can vary significantly.
2. *Branch predictor.* When out-of-order processors encounter a dynamic jump target, they can speculate which instructions will be executed next. Those instructions are then executed speculatively. This speculation will typically be based on previous control flow. The branch predictor keeps track of this control flow. When the actual control flow confirms the prediction, the speculatively executed instructions can be committed, resulting in a speedup. When the prediction turns out to be wrong, the speculatively executed instructions are flushed from the processor. Thus, the exe-

cution time of a program depends on the behavior of the branch predictor, which in turn depends on previous control flow.

3. *Return address stack*. This is an internal last-in first-out stack of return addresses for function calls. It mimics the operation of the program's stack for function calls. When the processor encounters a return operation, it uses this internal stack to speculatively determine the return address while waiting for the actual return address from the program stack. Because most function returns can be correctly predicted in this manner, this can speed up the execution [138].

1.6.2 Data flow

The execution time on most processors also depends on the data flow and the values of instruction operands.

Variable latency instructions Some instructions have an execution time that depends on their operands. Thus, there is a variable latency between the time at which the instruction starts to execute and its result is available. For example, some arithmetic operations such as multiplication and division can have variable execution times [13, 41]. This variation is caused by algorithms used in the processor's implementation for these operations that feature early exits.

Similar to the timing variation introduced by the instruction cache, processors also have data caches that can introduce data-dependent timing variation. The execution time of a memory access will depend on which cache has a local copy of the required data, if any.

Register dependencies Out-of-order processors will analyze the data dependencies of instructions, and use this information to reorder the execution of instructions and execute them concurrently where possible. For example, code in which each instruction's operand is the result of the previous instruction will execute slower than similar code with no such dependencies.

Resource contention Some processors allow instructions from concurrent processes and execution threads to execute concurrently on the same processor core. These threads then share the resources of one

core. When two threads require access to the same resource, the execution time of both programs will be higher when they execute concurrently. For example, when such a processor core only has a single integer multiplier, the execution time of a thread doing multiplications will depend on the number of multiplications done by other threads running concurrently [152]. An attacker measuring the execution time of a fixed number of multiplications thus leaks information on the number of multiplications executed by other threads.

1.7 Exploiting timing side channels

The discussed sources of timing variation can be used in different attacks to learn private information.

1.7.1 Measuring timing variation

Side channel attacks are often categorized by the level of access the attacker has to the target system. Three different kinds of attack are considered: (1) *time-driven attacks*, where the attacker can only measure the total execution time of a system, (2) *trace-driven attacks*, where the attacker produces a trace of some aspect (such as power) of the system by continuously monitoring it, and (3) *access-driven attacks*, where the attacker can access the system under attack and will measure the influence the target program has on the execution of the attacker's own code. Trace-driven attacks are typically not considered separately for timing side channels, since producing traces requires access to the target system. We briefly discuss how time-driven attacks and access-driven attacks can be performed.

Time-Driven Attacks A time-driven attack occurs when the attacker only needs to observe time behavior to perform the attack. He will either communicate directly with the target software, or will be able to observe the communications and their timing characteristics of another user communicating with the target software. By measuring the response time of the program under attack, the attacker knows the total execution time of this program under known inputs. This is a particular cause for concern for public-facing servers, to which the attacker can freely send and receive requests [30, 31], but also for smart cards,

in which a terminal can measure the time between sending a request to the smart card and receiving a result [86].

Access-Driven Attacks An attack is an access-driven attack when it requires an attacker to run his own code on the system running the program under attack. On the one hand, this gives greater power to the attacker to observe the program under attack, but on the other hand, this also restricts the attacker because he needs access to the system. Because of this access requirement, we need to consider how an attacker can gain access to the same machine. Apart from using a separate attack to force access to the target machine, there are generally two ways to get access to the same machine as the victim. The first is where the target code runs on a server to which the attacker has access as well. He can then attack (1) other users running code in parallel with the attacker's code, and (2) system code such as the kernel code that encrypts file system accesses [26]. The second way to get access to the same machine is when the victim runs his code inside a virtual machine, and the attacker gains access to a different virtual machine running on the same physical machine. This can be the case when the victim makes use of cloud computing services [125, 157].

Access-driven attacks exploit the fact that some hardware resources are shared between different processes, in particular between the attacked process, and a process under the control of the attacker. Shared resources used in attacks include data caches [23, 117, 119], instruction caches [2, 3], branch predictors [1], and shared functional units [152]. The attacker can measure how the target process interacts with the shared resources in three ways:

1. The attacker starts by measuring the execution time of the target program. Afterward, he makes a change to the shared resource, and then times the target program again. This allows the attacker to measure the influence the difference in states has on the target program [117]. This influence leaks information on how the private information interacts with the state. This requires multiple executions of the target program to get one measurement.
2. The attacker first sets the resource to a known state. After this, the target program executes. Finally, the attacker runs code that analyses the access time to the shared resource to determine the influence of the target program on the shared resource's state, for

example by measuring the number of cache misses [117]. Only a single run of the target program is needed for one measurement.

3. The attacker can run his own attack concurrently with the target code. He does this to get a trace of the shared resource's state over the course of a run of the entire target program. This way, the attacker can construct a trace of the target program. Such traces can be constructed at different granularities. For the finest possible granularity, the attacker can target systems running on processors with Simultaneous Multi-Threading support enabled [4], or he can abuse the scheduling behavior of operating systems [75] and hypervisors [157] to obtain traces that reveal the influence of almost every single instruction of the target program.

1.7.2 Recovering private information

While we cannot give an exhaustive discussion of how to use the obtained timing information in an attack, we give some examples as background information. This serves to show that an attacker can indeed recover this kind of information.

Most generic attacks use knowledge of the timing behavior of the target software, and correlate this with the observed timing behavior. For example, the first publicly known timing attack on RSA by Kocher [86] sequentially reconstructs the private exponent of certain implementations of modular exponentiation. As an example, take the implementation we described in Section 1.6, and assume that its modular multiplication has a variable execution time. The attack starts with recovering the leftmost bit. The attacker measures the execution time of the modular exponentiation on random inputs. He also knows the values of most variables in the first iteration: `result` is set to 1, `a` is a value supplied by the attacker, and `n` is part of the public key. Thus, he can model the execution time for the first iteration, with the private exponent's first bit set to either zero or one. The attacker subtracts the modeled execution times of the first iteration from the measured execution times. The variation will be lower when the exponent bit of the modeled execution matches the actual exponent bit used by the victim. By comparing the variances, the attacker can deduce the first bit. Using the first bit of the exponent, the attacker can then compute the values of the variables in the second iteration of the modular exponentiation. The attacker can thus sequentially discover all subsequent bits.

Such attacks can be generalized to adaptive side-channel attacks, where an attacker optimally chooses new inputs to the target program based on his prior measurements, as opposed to using random inputs. Using perfect knowledge of the implementation, an attacker can use information theory to describe the uncertainty of the private information. He can then decide which input to choose in each iteration to reduce this uncertainty as much as possible [88].

Another avenue of attack is to use the mathematical structure of the cryptographic algorithm under attack to speed up the attack. Take, for example, the Digital Signature Algorithm. This scheme uses secret, per-signature nonces in addition to a private key. Given a small number of bits from many such nonces, techniques exist to recover the private key [78, 120]. Some DSA implementations leak the number of leading zero bits in the nonces through the execution time. Thus, even though the implementation only leaks partial information, the mathematical structure allows an attacker to recover the entire private key [30].

For some cases of trace-driven attacks, patient attackers can manually study the traces, and annotate the different secret key-based operations on the traces [119]. However, in most cases this is not feasible because traces may be too noisy for a human to interpret easily, or the number of traces to perform the attack may be so high as to make a manual interpretation infeasible. Automatically identifying the internal states of the algorithm in regions of traces can be done using supervised classification techniques such as Hidden Markov Models [29, 157], Learning Vector Quantization [29], Support Vector Machines [157], etc.

1.8 Protecting against timing side channels

Since timing variation has been shown to be a problem of cryptographic algorithms, different solutions have been proposed to mitigate and remove timing variation from implementation.

New cryptographic algorithms Since the introduction of side channel attacks, new algorithms and protocols have been developed that are less vulnerable to the leakage of internal state, through timing and other such information. These leakage-resilient cryptography protocols [58] have the advantage that they are provably resistant to leaking

information through timing. Unfortunately they require the design of completely new algorithms.

Input blinding Sometimes, mathematical transformations can be applied to the structure of the cryptographic algorithm that make implementations more robust against timing side channels, instead of having to design new algorithms. This can be done by randomizing (or blinding) the input to the algorithm, and derandomizing the output afterward. Because the internal state of the program under attack is now randomized, the attacker has less information about the internal state, which makes attacking such implementations harder. For example, *signature blinding* has been used as a countermeasure against timing side channels in the case of RSA [86]. In this case, the message to be signed is multiplied with a randomly generated value before computing the RSA signature, and the effect of this random value is removed after computing the signature. While such techniques allow existing algorithms to be re-used in a safer way, they still require modifying the algorithm. Furthermore, blinded implementations are not necessarily provably secure, and in fact have been shown to be vulnerable in the case of RSA [17].

Hardware extensions Hardware vendors can implement specific algorithms in hardware. They can then ensure that the execution time is constant, and does not influence nor is influenced by shared resources. The AES-NI [74] extension to Intel's x86 instruction set provides software authors with a fast, side-channel-free hardware implementation of AES instructions.

More generally applicable hardware extensions can also be provided. For example, special cache architectures can be provided to lock access to parts of the cache that store cryptographic look-up tables, or to randomize the access time to certain sections of the cache [152, 153]. Similarly, some ARM CPUs have an extension that allows software to specify whether or not multiplications should be performed in constant time.

Timing-independent implementations Authors of cryptographic software can try to protect their code by rewriting it so that it no longer exhibits variable execution time. For example, table look-ups depending on the state of data caches can be removed and replaced

by constant-time instruction sequences that compute the required values [23]. Similarly, programs can be rewritten so that their control flow no longer depends on private information. Such techniques have been automated using source-to-source program rewriting techniques [108].

Rewriting programs to reduce their execution time variation will typically increase their execution time. However, software vendors can try to reduce the execution time variation of a program while still allowing for some variation to exist. Because the information gained by an attacker given timing measurements can be modeled [88], software vendors can thus trade off security versus run-time overhead depending on how much variation still remains [89]. One way to reduce the timing variation is by using *epochs* [14]. Epochs are time windows, and events such as input and output operations can only occur at the end of an epoch. Thus, the leaked information is related to the size of the epochs.

Adding noise Another possible mitigation strategy is to add noise to the execution time. When noise is added by the defender, the attacker needs to perform more measurements before being able to recover private information. This noise (or jitter) can either be introduced deliberately [87], or it might have been introduced inadvertently by the network over which the software's responses are routed [49]. However, this protection is not absolute.

When the attacker has access to the same machine as the target, his attack is limited by how precise his timing measurements are. Thus, instead of adding noise, a defender can restrict the precision of the timing information available to the attacker [79, 146, 106]. However, these techniques offer no protection for remote attackers who have an independent source of timing information.

1.9 Contributions

Both in attacks using code variation and attacks using timing variation, we note that variation can aid an attacker. When we want to protect against attacks that exploit variation, we can either try to increase variation, or we can try to decrease it in order to protect the users. The pros and cons of increasing or decreasing the timing variation can be described as follows:

1. *More timing variation.* As we already described, introducing more timing variation is a potential defense mechanism against timing side channel attacks. However, this will also introduce an overhead in execution time. Furthermore, the protection is not absolute: timing information will still leak through the measurements. Furthermore, access-driven attacks measure the influence of shared resources. Accesses to these resources thus have to be randomized as well. For example, even if the total execution time is randomized, the control flow during the cryptographic operations must be randomized to thwart such attacks. Otherwise, an attacker measuring the execution time of his own code will still detect the influence of the private information in the cryptographic code on his own accesses to the branch predictor, instruction cache, etc.
2. *Less timing variation.* We can reduce the timing variation by removing the control dependencies and data dependencies on private information. This protection is absolute in the sense that removing dependencies on private information will remove the dependent timing variation, which removes the attacker's source of information. The downside is that this can introduce a significant overhead in execution time.

We can similarly try to protect against patch-based attacks by increasing and decreasing the variation between the unpatched and patched program version. The pros and cons of those possibilities can be described as follows:

1. *Less code variation.* It is of course impossible to make the patched program identical to the original program. However, we can still try to reduce the variation to thwart the attacker. For example, we could try to protect against tools that only match instructions and not program data. In this case, we could try to make both programs have identical code, where only the data has changed. An extreme example of such a solution would be a protection mechanism where the program exists mainly of an emulator of a custom instruction set for which the actual instructions are located in the data section. These instructions will be encoded as custom instructions to be emulated. Even though a code matcher will completely match all program instructions, it will ignore the actual variation in the data.

2. *More code variation.* This is the strategy of software diversity. By artificially increasing code variation, the attacker will find it more difficult to identify a SCIM in the code.

In this PhD work, we have opted to defend against timing attacks by removing the control flow dependencies on private information, and to defend against patch-based attacks with software diversity. This PhD work consists of three contributions based on these defense mechanisms.

In Chapter 2 we introduce an abstract attacker model for patch-based attacks. We instantiate this abstract model using different real-world attack tools and heuristics. These models allow us to evaluate the effectiveness of different attack strategies in a patch-based attack, and show that such strategies indeed allow attackers to efficiently recover source-code induced mutations. Furthermore, we use these attack models to show that the Proteus diversification framework [7] can indeed be used as a defense strategy against attackers using real-world tools. This chapter reports on work that has also lead to a journal publication:

- Protecting your software updates
Bart Coppens, Koen De Bosschere and Bjorn De Sutter
In *IEEE Security & Privacy Magazine*,
Volume 11, Number 2, March-April 2013, pp. 47-54 [43].

Next, in Chapter 3 we show how we can use the attack models that make use of tools like BinDiff to improve the diversification strategy of Proteus. We introduce the diversification framework Glaucus, which uses the results of attack models as feedback. We show that with such a feedback-driven approach, the performance overhead of the diversified program is less than when using Proteus, while the attack effort is increased. This chapter reports on work that has also lead to a journal publication:

- Feedback-Driven Binary Code Diversification
Bart Coppens, Bjorn De Sutter and Jonas Maebe
In *ACM Transactions on Architecture and Code Optimization (TACO)*,
Volume 9, Issue 4, Article 24, January 2013 [44].

Finally, in Chapter 4 we show that we can use compiler transformations to protect effectively against control-flow based timing side chan-

nel attacks. Furthermore, we also introduce different program transformations that protect programs against some data-flow based timing side channel attacks. This leads to a timing side-channel aware compiler. This chapter reports on work that has lead to a conference paper:

- Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors
Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere and Bjorn De Sutter
In *IEEE Symposium on Security and Privacy (Oakland)*, 2009 [45].

In addition to these publications, I contributed to other publications that are either tangentially related, or not at all related to this PhD dissertation:

- Compiler mitigations for time attacks on modern x86 processors
Jeroen Van Cleemput, Bart Coppens and Bjorn De Sutter
In *ACM Transactions on Architecture and Code Optimization (TACO)*, Volume 8, Issue 4, Article 23, January 2012 [143].
- A Novel Obfuscation: Class Hierarchy Flattening
Christophe Foket, Bjorn De Sutter, Bart Coppens, and Koen De Bosschere
In *International Symposium on Foundations & Practice of Security*, 2012 [65].
- DNS Tunneling for Network Penetration
Daan Raman, Bjorn De Sutter, Bart Coppens, Stijn Volckaert, Koen De Bosschere, Pieter Danhieux and Erik Van Buggenhout
In *Annual International Conference on Information Security and Cryptology*, 2012 [124].
- An efficient algorithm for the generation of planar polycyclic hydrocarbons with a given boundary
Gunnar Brinkmann and Bart Coppens
In *Match—Communications in Mathematical and in Computer Chemistry*, Volume 62, Issue 1, 2009, pp. 209-220 [27].

Chapter 2

The effectiveness of variation against patch-based attacks

2.1 Introduction

In this chapter, we focus on attacks based on code variation. Given a patch for a program, an attacker can try to create an exploit for the unpatched program by studying the difference between the unpatched and patched program versions. All previous work on such attacks starts from the assumption that the number of mutations as reported by binary diffing tools is relatively small. The source-code induced mutations to the program stand out, and the challenging part of such attacks is generating an actual exploit.

Our approach to protect against such attacks consists of making the source-code induced mutations stand out less from translation-induced mutations by rewriting the entire program at a binary level. We build on software diversification, which is the introduction of artificial syntactic changes to generate different versions. Because of the additional syntactic changes, the attack requires more effort to find the vulnerability, which in turn delays the creation of the exploit. This gives users more time to apply the patch, and decreases the attacker's return on investment. Figure 2.1 shows the effect of delaying the attacker on the number of vulnerable users¹. We can distinguish two kinds of attacks:

¹This figure is stylized for clarity and is not based on any particular software. Some data exists on the speed with which users apply security patches. Frei et al. present

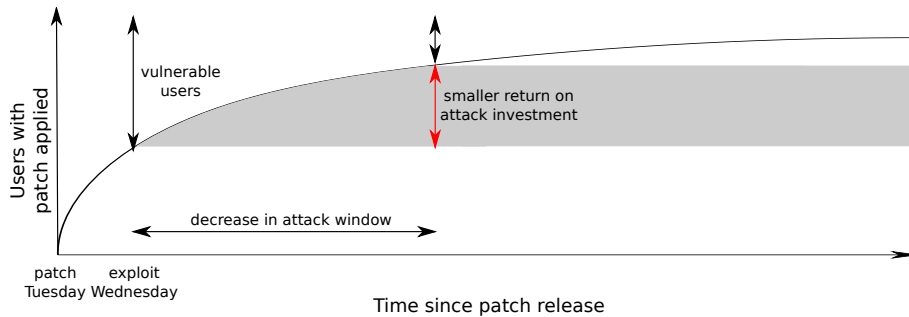


Figure 2.1: The attacker’s return on attack investment decreases when more users have applied the patch by the time the exploit is made.

non-persistent attacks and persistent attacks. In both cases, the attacker uses a vulnerability to gain control of the victim’s computer. With a non-persistent attack, the attacker only accesses the target computer for a limited amount of time. For example, if the attacker only wants to steal personal information, it is not important that the attacker loses control over the victim once the victim restarts his computer or applies the security patch. In such cases, when the attacker’s window of opportunity is decreased, his smaller return on investment is represented by the number of victims he can no longer steal information from. This number of lost potential victims is shown by the red vertical arrow in the figure. With a persistent attack, however, the attacker wants long-term control over the victim’s computer. This control needs to persist across reboots and installed updates. For example, when an attacker wants to use the target computer in a botnet, long-term control is required. When the attack is delayed in such cases, the attacker loses *future use* of the potential victims who have installed the security update in the meantime. This loss for the attacker is represented by the shaded area for the case where the attacker persists even after the security update is installed.

Software diversity has already been applied for defensive purposes by Anckaert to hide similarities between programs from diffing tools by making the patched program look dissimilar from the original program [7]. He used his Proteus software diversification framework to evaluate his custom binary diffing framework. Proteus was not evaluated on tools that actual attackers use. It was evaluated on differently diversified instances of the same program rather than on patches.

and analyze such data for the Firefox and Opera web browsers [69].

The question remains whether or not software diversity can protect users against a patch-based attack with real-world attack tools. In order to answer this question, we first formally define which binary instructions correspond to the the source-code induced mutations of a patch (Section 2.2). Since we will use real-world attack tools, we introduce a novel abstract model of how these tools may be used by attackers to find SCIMs, and offer concrete instantiations of these models (Section 2.3). These models can be used to compare both different attack strategies, and different protection strategies. We evaluated these models on programs diversified using the Proteus diversification framework, which is described in Section 2.4. Our attack models then allow us to show that existing attack tools can indeed significantly aid patch-based attacks on unprotected binaries by pointing attackers to a very limited set of changed instructions containing the SCIMs. Furthermore, we then show that effectiveness of patch-based attacks are indeed thwarted when the binaries have been diversified with the Proteus framework (Section 2.5).

2.2 SCIMs and TIMs

The goal of the attacker is to find source-code induced mutations at the binary code level. In Chapter 1 we already used this term informally, but we never defined it. In order to evaluate attack tools, we need to know which altered instructions an attacker *should* find. Thus, we need to give a definition of SCIMs and TIMs.

An obvious definition would be to require SCIMs to be semantic changes in the program, i.e., the changes to instructions that give rise to a change in the input/output behavior of the program. However, this definition would be too strict. We already described how the timing behavior of a program can leak information. A fix for such a vulnerability would not change the input/output behavior of the program at all, while it is still a fix for an important vulnerability. Similarly, some programs use algorithms with problematic worst-case behavior. Attackers can exploit such worst-case behavior in denial-of-service attacks [85]. Fixes for such issues again only affect the timing behavior of the program.

Instead, we will base our definition source-code induced mutations for binary programs on the effect of changes to source code.

Definition. *The source code mutations of a source code patch are the program statements and definitions that are changed by the patch.*

These source code mutations can be identified by the textual representation of the source patch. In order to produce a binary program, the compiler front-end will parse the source code into an Abstract Syntax Tree (AST). This AST will then be transformed into an intermediate representation, on which the different compiler analyses and transformations will run. The end result of these transformations will then be used by the compiler back-end to generate the binary code [110]. To do so, we track source changes through the compilation process:

Definition. *Given a set of altered source code statements or definitions, we can tag the nodes that correspond with these statements or definitions in the AST. These tags can be propagated through the compilation process to the binary code generation. All compiler transformations will update and create new tags when refining or duplicating nodes that have been tagged. We call the bytes of the target program that have been tagged the corresponding bytes of the set of altered source code statements or definitions.*

This means that if, for example, a tagged function call is inlined, the compiler transformations will tag the duplicated code as well. When the tagged function causes changes in, for example, register allocation, such changes will not be tagged.

This allows us to define the binary source-code induced mutations as follows:

Definition. *The binary source-code induced mutations of a source code patch are the corresponding bytes of the source code mutations of this patch.*

We can then define translation-induced mutations by comparing them to source-code induced mutations:

Definition. *The translation-induced mutations of a source patch are all mutations of a program that are not contained in the set of source-code induced mutations for this source code patch.*

For simplicity's sake, when a SCIM or a TIM is located in an instruction, we will refer to the entire instruction when referring to this mutation.

This enables us to specify which binary code instructions correspond to source code changes. When a source code patch set consists

of multiple patches, we can, at the source level, partition the different changes and we can tag them differently, allowing us to distinguish between different subsets of a single patch set. We can also use different tags when a software patch fixes multiple issues, some of which are security-related, and some of which are not.

Such tags could be generated by existing compilers similarly to how debugging information is generated. Debugging information typically contains a mapping of source code lines to instructions. In this work, we have not implemented this in a compiler. We manually inspected the patched binary to determine the instructions corresponding to the changed source code lines.

2.3 Heuristic attack model

The attacker needs to find the SCIMs as fast as possible to have the largest possible attack window. As discussed in Chapter 1, an attacker will try to separate the SCIMs from the TIMs using binary diffing tools before trying to make an exploit. He will rely on the abstractions of many syntactic changes provided by the diffing tools to focus on the SCIMs. The attacker's work flow is shown in Figure 2.2. The attacker starts by applying the binary patch to the unpatched program, obtaining the patched program. He then uses a binary diffing tool to find differences between the unpatched and patched program versions. The code fragments that the diffing tool reports as changed or unmatched are then inspected by the attacker. When there are multiple code fragments that need inspection, the attacker will try to prioritize them in order to find the fragments containing the SCIM as quickly as possible. When he has found the SCIM corresponding to a vulnerability fix amongst these code fragments, he can construct an exploit.

The goal of this chapter is to prove the effectiveness of diversity when it is used to reduce the effectiveness of the diffing tools. We will use real attack tools to show this. We will prove the effectiveness by quantifying the "delay" incurred by attackers when software diversity is applied. Ideally, we would perform a social study in which we give different attackers either the original program and the software patch, or diversified versions of the programs. By measuring how long it takes the different attackers to create a working exploit (if they are able to create one), we can measure the attack delay. However, such experiments are time consuming, costly, and hard to repeat when new countermea-

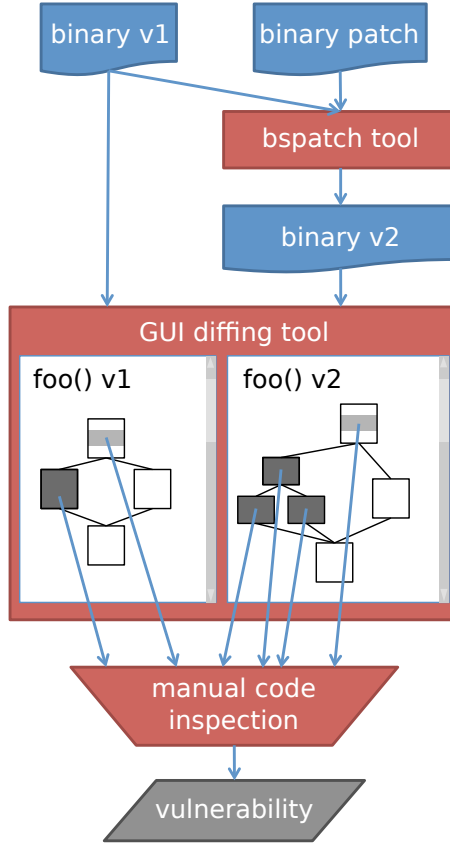


Figure 2.2: Attacker’s tool flow to find SCIMs in a binary patch.

asures need to be evaluated [34, 35, 142]. Furthermore, every attacker can use a different combination of tools, prioritization heuristics, and insights to find the SCIMs. As an alternative to performing studies with human subjects, we model the way in which an attacker uses his attack tools. We define an abstract framework to model an attacker, and instantiate this model with different combinations of attack tools and prioritization strategies. These models can then be evaluated and compared.

2.3.1 A framework for attack models

The attack models we consider will be concrete instantiations of the following abstract attack model:

Definition. Given unpatched program version U , patched program version P , and a prioritization strategy π , the attack model $\mu_\pi(U, P)$ is the set of instructions that obtain the highest priority when applying strategy π on the pair of programs (U, P) .

We assume that there is no further subdivision of this set. So in the case of manual exploit generation, we envision that an attacker can randomly select instructions to analyze from this set, or, as in the case of automated fuzzing, target all instructions in this set. Thus, for a prioritization strategy π , we could use the cardinality of this set, $|\mu_\pi(U, P)|$, as a proxy for the attack effort.

Note that in the case of an automated attack, the attack cost depends on the number of instructions that need to be automatically evaluated for creating an exploit. In that case, the cost is a computational one. Based on the normalized set size, we can report how useful a strategy has been for an attacker. First, we present the pruning rate:

Definition. The pruning rate of the attack model $\mu_\pi(U, P)$ is

$$1 - \frac{|\mu_\pi(U, P)|}{|P|}.$$

As such, it directly relates, linearly or not, to the attacker's effort and time. The higher the pruning rate, the lower the attack time. This is different from the more commonly used precision, because the pruning rate focuses more on the effort for the attacker to analyze the returned instructions.

An attacker cannot evaluate the effectiveness of his attack strategy until after he has found the SCIMs. However, in order to evaluate and compare different concrete attack models, we can use binary patches for which we know the ground truth. We start from benchmarks that we applied a binary patch to. For these benchmarks, we know the ground truth, which is the set of binary instructions corresponding to the source code patch. The ground truth for transforming program version U into program version P is denoted by $\tau_{\text{patch}}(U, P)$. We can then express the success of the strategy using the recall rate:

Definition. The recall rate of the attack model $\mu_\pi(U, P)$ is

$$\frac{|\mu_\pi(U, P) \cap \tau_{\text{patch}}(U, P)|}{|\tau_{\text{patch}}(U, P)|}.$$

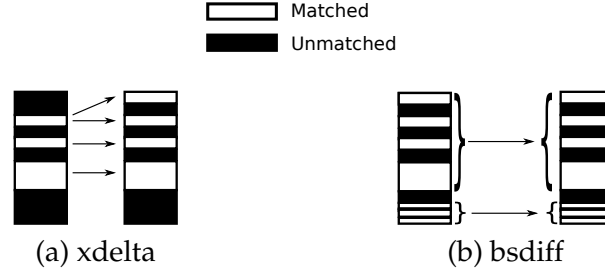


Figure 2.3: Schematic overview of the attack tools operating on byte sequences.

If the recall rate is zero, none of the SCIMs are contained in the list of instructions in the attack model and the attack will obviously fail.

We will evaluate different combinations of attack tools and prioritization strategies on the same pair of unpatched-patched programs (U, P). It is possible that these tools only return a set of matched functions, or of matched basic blocks. In such cases, we consider $\mu_\pi(U, P)$ to be all instructions inside those basic blocks and functions. Note that we do not require that these instructions are assigned to the correct basic block or functions, or even that the instructions returned are actually part of the program. The attacker can only use the information returned by the attack strategy, and will rely on it to find the vulnerability.

With this attack model framework, we discuss the different strategies π which combine attack tools with heuristic pruning strategies. We will drop the arguments U and P from the attack model and ground truth when they are clear from the context.

2.3.2 Binary diffing tools

The attacker can first prioritize his search based on raw diffing results. These diffing results are a set of instructions that have been matched between the program versions. With this information, the attacker can use different heuristics to further prioritize his search.

In general, an attacker can try to prioritize the unmatched code fragments; these will most likely contain SCIMs.

We first discuss how an attacker could use and interpret the raw results returned by these diffing tools, and how we can capture this attacker behavior.

Byte-level diffing tools

Both `xdelta` and `bsdiff` create binary patches by comparing the patched program to the original program. The resulting patches are binary script files with instructions on how to transform the original binary into the patched binary. The patched program then consists of bytes copied directly from the original program, and bytes introduced by the patch script. An attacker can tweak these tools to output a list of bytes in the patched program that were not copied directly from the original program. He will prioritize these bytes over the ones that could be copied, and will use a disassembler or debugger to see to which instructions these bytes belong.

Figure 2.3 visualizes the information attackers can extract from the patches generated by `xdelta` and `bsdiff`. Copied fragments are shown in white. The arrows indicate the origin of copied regions in the patched program. The fragments shown in black represent code fragments that are not copied from the original binary, but have been stored in the patch file itself. Additionally, `bsdiff` supports copy-and-patch operations that specify how a large chunk of bytes can be copied and then patched. While these copy-and-patch operations were clearly designed to handle small translation-induced mutations efficiently, their use is not limited to TIMs. Hence the small patches applied to copied bytes still have to be analyzed manually by the attacker. Figure 2.3 (b) hence depicts them in black.

Byte-level tools consider bytes without interpreting them. When bytes are copied by such tools, this only implies that the source and destination bytes are identical. There need not be any semantic relation between the source and destination. For example, when a certain byte sequence is repeated in different contexts throughout the binary, byte-level tools can pick one such sequence and copy this to all destinations. This is indicated in the figure by a single block having multiple outgoing edges. This happens frequently, for instance with function prologues and epilogues [53]. This clearly reduces the usefulness of such tools.

It should also be noted that these tools try to optimize the size of the patch. So they may prefer to produce one insert operation of a large block over producing a sequence of smaller copy and insert operations, thus hiding that there are actually many similarities between some code fragments. This further reduces the usefulness of such tools in most situations.

While at first sight the attacker's search space has been reduced to only the parts shown in black, we note that the mapping between identical white parts is not necessarily correct or useful. For example, when the code can be mutated in such a way that the mutated code already occurs in the original binary. The patch tool can thus return to the attacker that this code has not mutated at all in the patched binary. So in that case the tool turns the attacker away from the interesting code. On the other hand, because of the level of abstraction of these tools, most syntactic mutations are not abstracted away. This potentially increases the attack effort significantly.

Even so, these tools help the attacker prioritize code fragments, by pointing out code fragments in the patched program that do not correspond to any fragment in the original program. The attacker could first focus on these bytes, before trying to analyze more code.

We can model attacks using byte-level diffing tools by identifying the bytes from the target program that have been copied from the original program. We can prune these identified bytes. Since these byte-level tools operate on the binary level, rather than at the assembly level, we envision that the attacker will first try to disassemble the binary to determine which instructions were added or modified. We model an attacker prioritizing exactly the instructions that have changed according to the binary patch tools (which is the set of code fragments shown in black in the figure) as μ_{bsdifff} and μ_{xdelta} .

IDA Pro-based Tools

The other diffing tools we consider are plug-ins for the IDA Pro disassembler. They give the attacker a list of code fragments that have been matched in both program versions. These programs help the attacker by differentiating between code fragments that form an *exact* match, those that form a *near* match, and those fragments that do not match at all at the level of abstraction of the tool.

As opposed to the binary patch tools, these attack tools all have a Graphical User Interface (GUI) through which the attacker will study and prioritize code fragments. We therefore have to model a prioritization strategy on the GUI.

First, the attacker will run IDA Pro on the binaries to disassemble the code. The attacker may instruct IDA Pro to disassemble code that was incorrectly classified as data. Then, the attacker will run a diffing

tool. This tool will return a list of matched function pairs, and a list of unmatched functions. Some attack tools offer the possibility to show potentially matched functions side by side, highlighting the unmatched BBLs. We envision that the attacker will study each of the unmatched code fragments one by one until he has studied all unmatched code fragments. For this, we can prioritize the unmatched code using different strategies.

To model these attacks, we identify the instructions that are matched according to the diffing tools². In the cases where the diffing tool only returns matches of whole functions or BBLs, rather than individual instructions, we consider all the instructions contained in the unmatched functions or BBLs. All instructions that are not matched by the diffing tools are instructions that the attacker will have to analyze or prioritize himself.

All tools that use IDA Pro as a base for disassembly and function detection will have similar problems when IDA Pro fails at this task. Code fragments that do not belong to a function according to IDA Pro will not get matched by any tool, unless the tool adds additional heuristics to group functionless code fragments. The tools that use IDA Pro can also make abstraction of references to other code fragments: they are automatically aware of when a change to an instruction is in an offset.

Patchdiff2 The patchdiff2 IDA Pro plug-in only returns matches at the function level. Figure 2.4 shows a screenshot of patchdiff2. It reports functions either as being identical, being similar, or being unmatched. These are grouped in three different lists. The only information reported by patchdiff2 is the function addresses of the matching functions, and the computed checksums of the functions. As can be seen in the Figure, an attacker must open the two binaries in IDA Pro side-by-side himself and manually navigate to the functions he wants to study further.

An attacker can then study the functions that have not been matched at all, after which he can investigate similar functions. When trying to find the SCIMs using this information, he is left to analyze the difference between entire functions himself. The attacker can prioritize his

²To be able to analyze the diffing results automatically, we extended the plug-ins to dump the diffing information to a file when this functionality was not present in the tool.

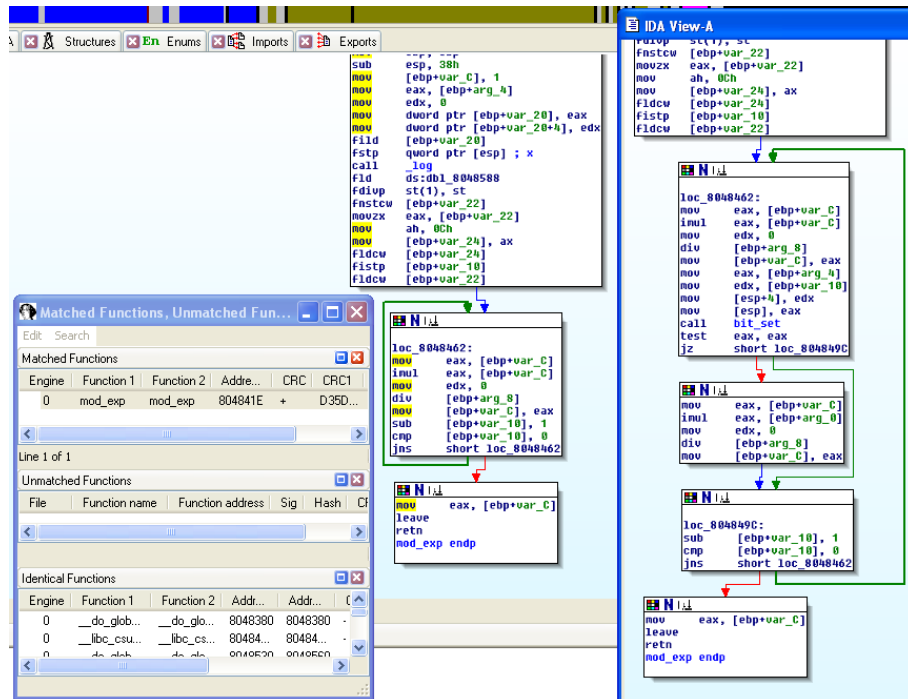


Figure 2.4: Screenshot of the patchdiff2 diffing tool.

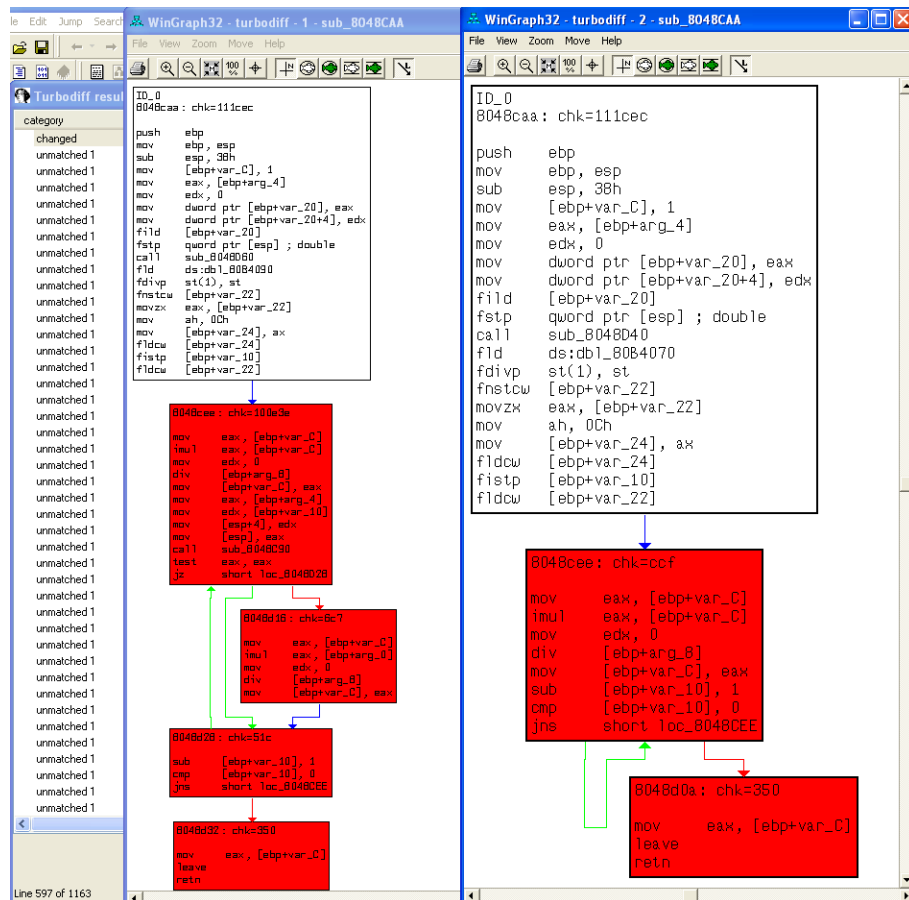


Figure 2.5: Screenshot of the TurboDiff diffing tool.

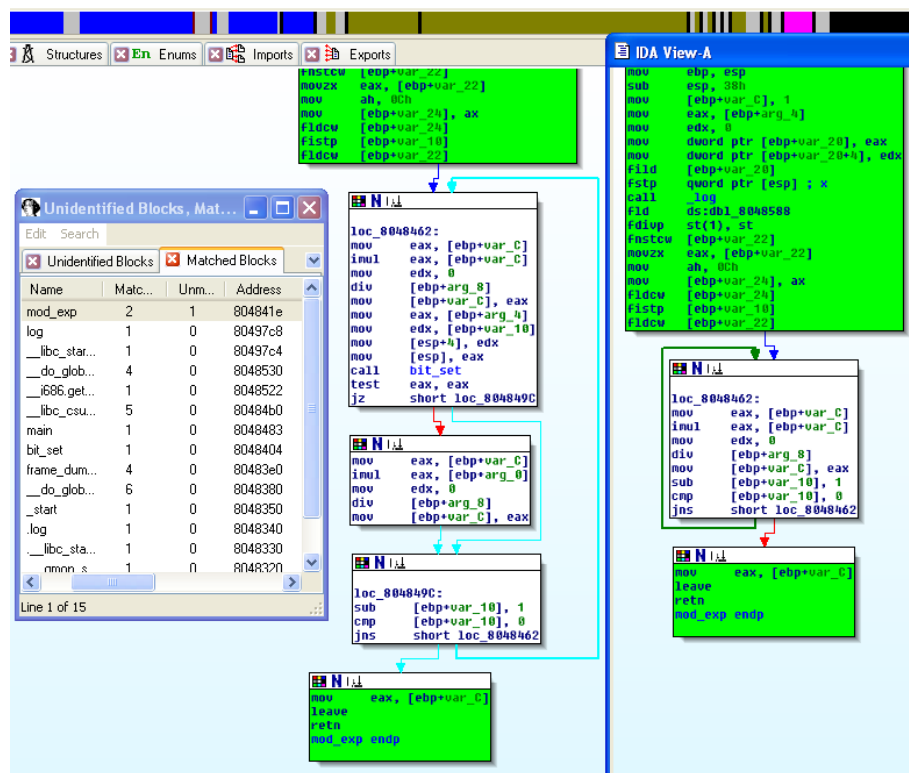


Figure 2.6: Screenshot of the BinaryDiffer diffing tool.

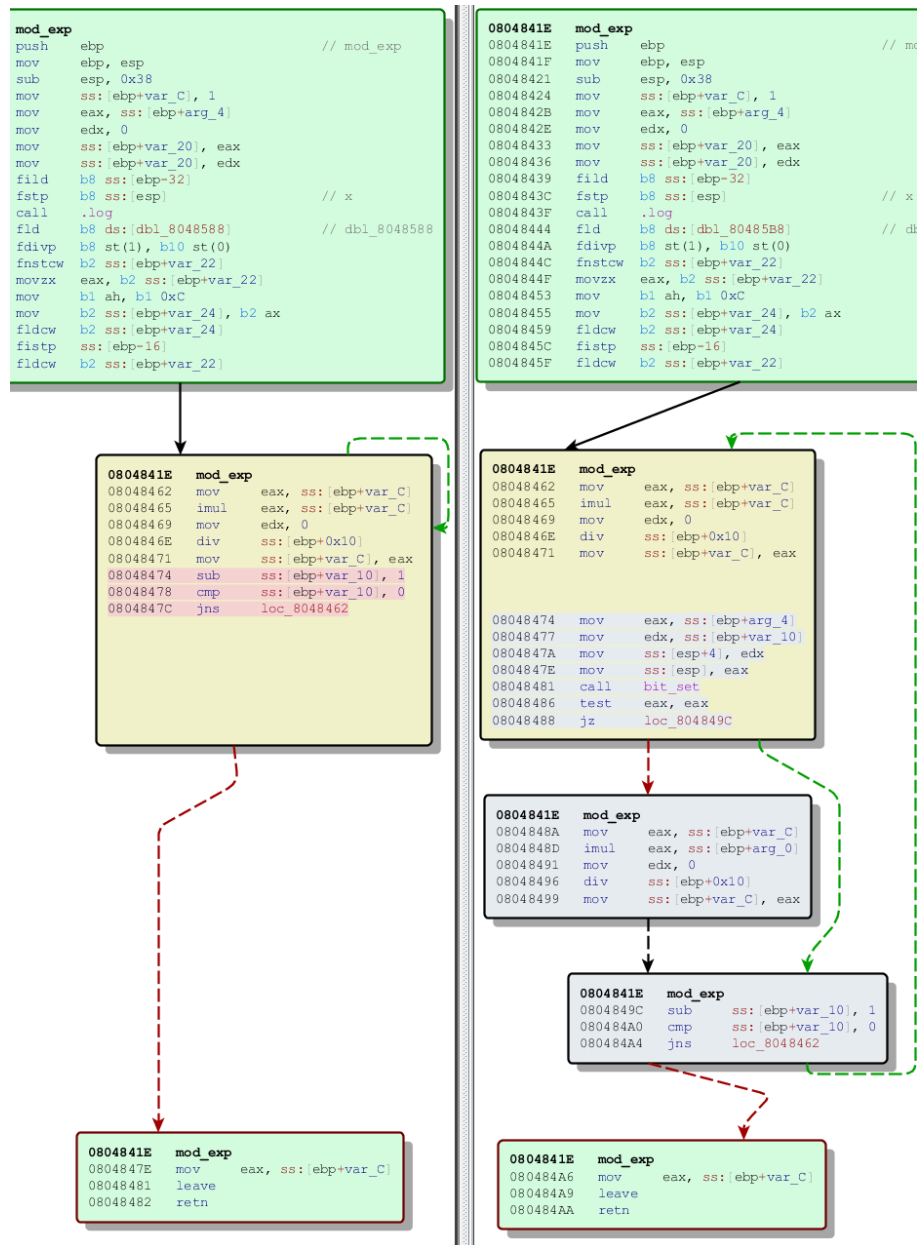


Figure 2.7: Screenshot of the BinDiff diffing tool.

results depending on the match strength of the matching functions. We model two different prioritization strategies: $\mu_{\text{patchdiff2: identical+similar}}$, where we use patchdiff2 and the attacker ignores identical and similar functions, and $\mu_{\text{patchdiff2: identical}}$, where we use patchdiff2 and the attacker only ignores the instructions from identical functions.

BinaryDiffer and TurboDiff BinaryDiffer and TurboDiff are two binary diffing plug-ins for IDA Pro that both give information about matches between different basic blocks. Figure 2.5 is a screenshot of TurboDiff in operation. We see that TurboDiff can show the user a colored representation of matching functions in a separate viewer, where red BBLs are unmatched. The operation of BinaryDiffer is shown in Figure 2.6. Like patchdiff2, it requires the attacker to open two instances of IDA Pro at the same time, each with one version of the binary program. Like TurboDiff, it colors the matched BBLs differently, but it does so in IDA Pro’s GUI, rather than in an external viewer.

BinaryDiffer reports pairs of corresponding, identical basic block, pairs of corresponding, very similar but non-identical blocks, and pairs of corresponding but more heavily mutated basic blocks. TurboDiff reports only pairs of corresponding, identical basic block, and pairs of corresponding, very similar but non-identical blocks.

The attacker can prioritize those basic blocks that are not reported to be similar by the tools. If he learns no information on the SCIMs from studying those basic blocks, he can look at the basic blocks that have been identified as being similar, but not identical, etc.

Thus, we have attack models based on how similar the matched BBLs are according to the attack tools. For BinaryDiffer we have the models $\mu_{\text{BinaryDiffer: identical+similar+heavily mutated}}$, $\mu_{\text{BinaryDiffer: identical+similar}}$ and $\mu_{\text{BinaryDiffer: identical}}$. For TurboDiff we have $\mu_{\text{TurboDiff: identical+similar}}$ and $\mu_{\text{TurboDiff: identical}}$.

BinDiff BinDiff is a binary diffing plug-in for IDA Pro that reports matching instructions. This is shown in Figure 2.7. It has a separate viewer that colors individual instructions in matched functions according to whether or not they are matched with an instruction in the other binary. Furthermore, the instructions and BBLs are laid out such that they can easily be compared visually.

For BinDiff, instructions either match, or they do not match. The attacker can thus focus first on the instructions that do not match accord-

ing to BinDiff. We consider an attacker who only focuses on unmatched instructions: $\mu_{\text{BinDiff: matched}}$.

2.3.3 Additional prioritization heuristics

On top of the raw tool results, the attacker can use additional prioritization heuristics. We model these as follows.

Byte-level tools

The abstraction level at which the byte-level tools operate is significantly lower than that at which the IDA Pro-based tools operate. The latter ones can use disassembled instructions and can base their diffing results on more abstract representations, while the former will report more TIMs. An attacker using byte-level tools such as bspatch and xdelta can try to prioritize the byte-level results, removing from the byte-level output some of the changes that are not reported by the IDA Pro-based tools.

Keep only instructions Since the byte-level tools make no distinction between changes in code segments, and changes in data segments, the attacker will also have to analyze the changes to data. A first prioritization heuristic is that the attacker only looks at changed instructions.

We implement this by basing the results of our models only on the changes that happen in the code sections of the patched program. Because it is so basic, we already incorporated this prioritization strategy in our attack models μ_{bsdiff} and μ_{xdelta} . However, attack models can be designed so that they also include changed data.

Prune mutations limited to immediate operands of instructions As explained in Chapter 1, when code is moved around in a program's memory due to a patch, both absolute and relative addresses encoded in the binary code and in the data of a program get modified. The number of changed offsets is typically much higher than the number of instructions constituting the SCIM. Thus, the attacker can prioritize code fragments by pruning instructions with only changed offsets.

We prune the instructions where the only changes are to the immediate bytes of the instruction. We use a linear sweep disassembler

to decode the instructions and to determine where in the instructions the immediate bytes of the instructions lie. We can extend a concrete attack model μ_π with this heuristic, resulting in the attack model $\mu_\pi \wedge \text{prune changed immediates}$.

IDA Pro-based tools

For the diffing tools based on the IDA Pro disassembler, the quality of the results not only depends on the implementation of the diffing tool, but also on the quality of the IDA Pro recursive descent disassembler. To improve the quality of the results, we envision that an attacker could use custom heuristics. We now present different heuristics we believe might aid an attacker.

Extending IDA disassembly As mentioned before, IDA Pro’s automatic recursive descent disassembly process does not automatically detect all code in a program. In its interactive operation mode, an attacker can indicate additional locations in the binary code that IDA Pro should disassemble.

To automate this, we implemented a plug-in that mimics such an attacker. This plug-in is invoked in between the standard IDA Pro disassembler and the diffing plug-ins. It extends the standard disassembler by forcing IDA Pro to disassemble all parts of the binary not yet disassembled and not determined to be data. The attack model μ_π after extending IDA’s disassembly process is denoted by $\mu_\pi \wedge \text{extend IDA}$.

Considering code not disassembled as irrelevant Another potential solution to the problem of IDA Pro not automatically finding all functions is to ignore such code altogether and explicitly filter these instructions from those the attacker has to consider manually. By ignoring these instructions, as opposed to trying to analyze them, the attacker runs the risk of removing the SCIMs from his consideration. We denote these attack models by $\mu_\pi \wedge \text{only disassembled}$.

Tool-specific CFG pattern pruning Some tools do not use certain parts of the control flow graphs to determine the matching code fragments. When certain parts of the CFG are unmatched, the attacker would have to look for these unmatched code fragments. Again, he

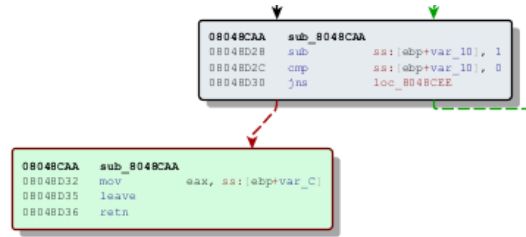


Figure 2.8: The attacker will also study matched code fragments adjacent to unmatched code fragments.

can prioritize the changes by specifically ignoring such unmatched code fragments.

We considered three specific code patterns that are not present in the output of BinDiff. These are no-operation instructions, basic blocks containing only a direct control transfer instruction, and basic blocks that are unreachable from function entry points. We denote these attack models by $\mu_{\pi \wedge \text{prune patterns}}$.

Generic prioritization strategies

Finally, we discuss prioritizing two heuristics that apply to both the results of the byte-level tools, and the IDA Pro-based tools.

Only executed code Attackers can prioritize code that gets executed on typical inputs. Code that gets executed is more easily studied by the attacker by using a debugger and tracing tools. Furthermore, such code will be easier to trigger, which may facilitate generating an exploit.

We simulate this by tracing the program execution on test inputs, and prioritizing the executed instructions. We use the Diota³ dynamic instrumentation framework to trace the programs. Such attack models are denoted by $\mu_{\pi \wedge \text{executed}}$.

Expand the window of instructions To study the results of the diffing tools, an attacker will use a disassembler and debugger. This disassembler not only shows the unmatched code, but also the code around it. Thus, the attacker will also view and study matched code fragments around unmatched code. For example, in Figure 2.8, even though the

³<http://www.elis.ugent.be/diota>

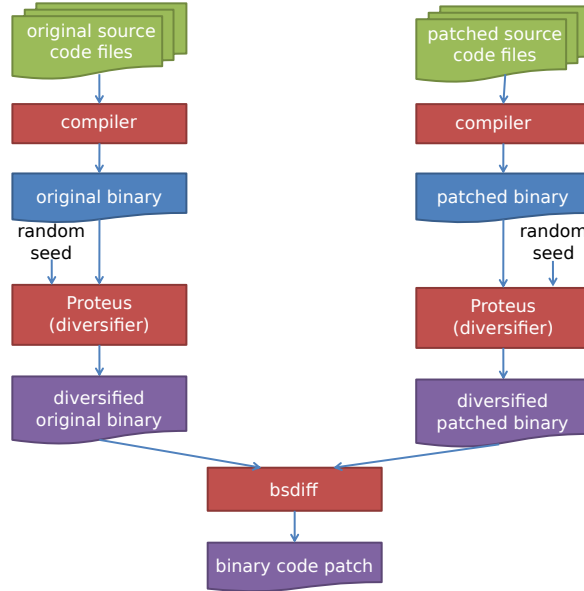


Figure 2.9: Tool flow when distributing a patched program version using the Proteus diversification framework

mov instruction is matched by BinDiff, the attacker will still analyze it when analyzing the code in the unmatched BBL that precedes it. An experienced attacker will in practice notice almost immediately that one of the neighboring instructions is also relevant.

However, when computing the recall rate for our attacker model, we only consider the unmatched instructions as reported by the diffing tools. In order to make up for this problem, and to model an attacker with a GUI, we include an additional type of “heuristic” that extends the code window considered relevant by the tool with its immediate neighbors. When we include this heuristic in an attack model, which is denoted by $\mu_{\pi \wedge \text{expand window}}$. Different window sizes can be used for different models. In the models we evaluated, we only expanded the window when adjacent instructions are indeed part of a SCIM. Different expansion strategies could be used instead of this optimistic window expansion.

2.4 Diversification as mitigation strategy

As explained earlier, software diversity has been proposed as a mitigation against several collusion-based attacks, including Exploit Wednesday attacks. Our goal is to prove that this is indeed an effective protection against real-world attacks. We do this by evaluating different concrete attack models both on undiversified and diversified program pairs. To study the impact of diversification, we used the diversifier Proteus [7] that comes with the Diablo link-time rewriting framework⁴ [51]. Figure 2.9 shows the proposed tool flow of using the Proteus diversifier. A software vendor using Proteus starts by compiling and linking the source code of the programs to binary code. Proteus is then run on these binaries before the resulting diversified binaries are distributed to end users.

Proteus' underlying Diablo framework supports a number of standard code generation, optimization and obfuscation techniques, but rather than optimizing towards a performance or software protection objective, Proteus applies them in a stochastic manner using a pseudo-random number generator (PRNG). The order in which the different transformations are applied is fixed. Different versions of a binary can be generated simply by feeding the PRNG with different seeds. To trade off the level of diversification with the overhead introduced by the stochastic application of transformations, the user can select the probabilities with which transformations are applied. These transformations will only be applied when the necessary preconditions are met, ensuring their correctness. There is a further attempt at limiting the overhead by ensuring that high-overhead transformations are only applied to cold code, i.e., code that is executed relatively little according to collected profile information.

2.4.1 Diversifying transformations in Proteus

The transformations supported by the Proteus diversification framework can be grouped into three categories: compiler optimizations, code generation techniques, and obfuscation techniques.

⁴<http://diablo.elis.ugent.be/>

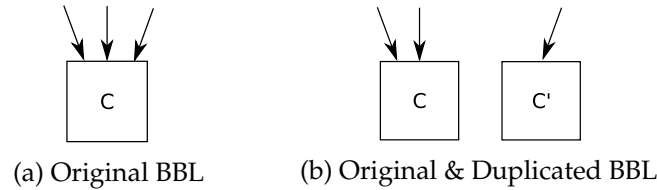


Figure 2.10: An example of tail duplication for basic blocks. The transformation duplicates the BBL, and lets one of the incoming edges of the original BBL point to the duplicate.

Compiler optimizations

Some transformations are typically used in a compiler for optimizing the generated code for typical compilation objectives such as execution time and code size.

Function inlining Function inlining replaces function calls with copies of the body of the called function [110]. This is useful to optimize programs for speed, because it removes the call overhead, and it allows analyzing and optimizing of the called function in the context of the caller without having to use intra-procedural analyses. A potential drawback of this method is that it can increase code size significantly.

Code factoring Code factoring is the opposite operation of function inlining for functions; it merges identical code fragments into a single copy [55]. Proteus can apply this transformation on entire *functions*, on *function epilogues* and on *basic blocks*, which become a function.

Tail duplication Tail duplication duplicates a basic block with multiple incoming edges. This transformation is illustrated in Figure 2.10. When a basic block has multiple incoming edges, we can duplicate the basic block and redirect some of the incoming edges in the CFG to the duplicate block.

Compiler code generation techniques

Compilers transform IR into machine code with code generation techniques. The goal of these transformations is typically to produce instruction sequences that are as fast or as small as possible. These trans-

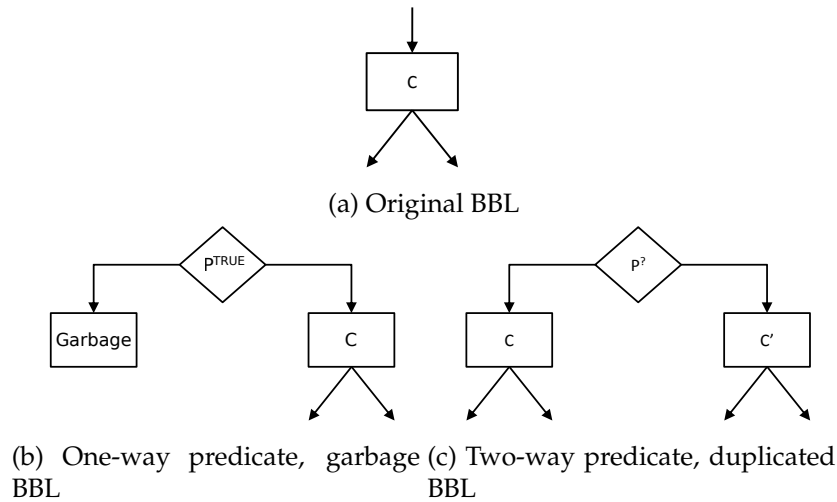


Figure 2.11: Predicating a basic block by a two-way opaque predicate

formations include changing the positions of basic block chains (i.e., sequences of basic blocks chained together in fall-through paths), the reordering of the instructions in basic blocks insofar as their dependencies allow this, and instruction selection. For diversity, we select random alternatives for the code that needs to be generated, instead of the optimal ones.

Obfuscation techniques

Obfuscation techniques aim to make the binary code less understandable, and harder to reverse engineer. Their primary goal is to hide the functionality or location of the code. Proteus uses these techniques for their ability to make code look dissimilar.

One-way opaque predicates One-way opaque predicates are conditions that always evaluate to the same binary value during program execution, but for which it is hard to prove this property statically [42]. This value is then used as the predicate of a conditional jump instruction. This jump instruction has two outgoing edges in the CFG: one edge for when the predicate is true, one for when the predicate is false. The edge in the CFG that corresponds to the value that is never gen-

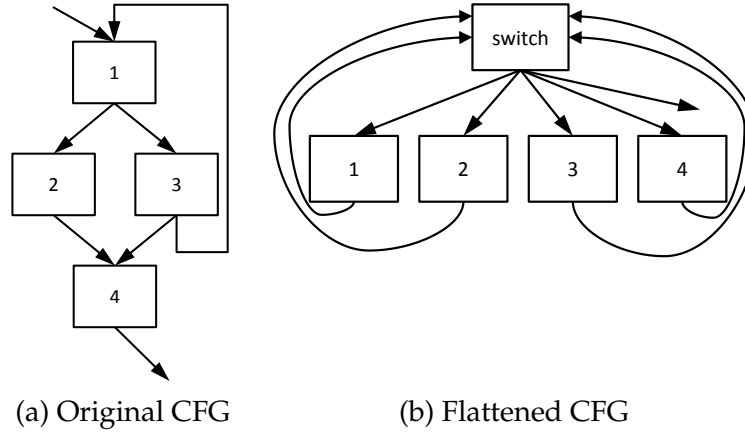


Figure 2.12: Control flow flattening

erated by the program can point to anywhere in the program without affecting the program’s semantics. In Proteus, this code path simply points to a mutated copy of the basic block to which the opaque predicate transformation was applied. This transformation is illustrated in Figure 2.11 (b).

Proteus injects one-way opaque predicates taken from a small library of such predicates [11, 42].

Two-way opaque predicates Two-way opaque predicates are predicates that can evaluate to both true and false during the program execution [42]. When transforming a basic block with this transformation, this basic block is duplicated and added to the control flow graph so that regardless of the condition, the required code is executed. This transformation is illustrated in Figure 2.11 (c).

Control flow flattening Control flow flattening is a transformation that transforms a CFG into one in which every basic block from the original CFG has the same predecessor, the so-called switch block [149]. In the transformed CFG, the successor of a BBL is selected by letting a BBL communicate its successor to the switch block using a variable. Figure 2.12 illustrates this transformation. In the CFG of Figure 2.12 (a), the possible control flow is immediately obvious. This is no longer the case for the flattened CFG in Figure 2.12 (b). The switch block transfers control to one of its successor BBLs, depending on the value of the

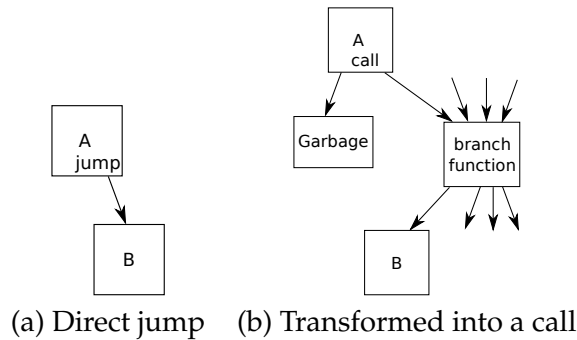


Figure 2.13: Schematic representation of a branch function

variable. Studying only the static CFG could give the impression that, all BBLs could, through the switch block, be successors of one another.

Proteus only applies this transformation to a function's entire CFG.

Branch function redirection Branch functions are functions that instead of returning to the caller, immediately transfer control to a different address. This is used to transform direct jumps into function calls and indirect jumps [98]. Garbage code is inserted in the fall-through path of the call to the branch function. This garbage code is chosen such that it ends half-way an x86 instruction. Linear disassemblers will thus disassemble this garbage code, and continue disassembling the garbage into the next actual basic block, thwarting analyses that build on this disassembly. This transformation is illustrated in Figure 2.13.

Proteus transforms direct jump instructions into two push instructions followed by a call to a single branch function. One of the pushes is to store the flags register on the stack, because this register is modified by the branch function. The other push is an offset that is used by the branch function to compute the return address of the branch function. Garbage data is added in the fall-through path of the added call.

Application of the transformations

As explained earlier, Proteus will stochastically apply these transformations using probabilities that can be set on a per-transformation basis. Table 2.1 shows some of the selected probabilities that Proteus has been used with previously to evaluate Anckaert's matching framework [7].

There are multiple reasons to apply these particular transforma-

Table 2.1: Different settings of the diversity system

| Transformation | mapping | CFG change | CG change | p_1 | p_3 |
|--|---------|------------|-----------|-------|-------|
| <i>Compiler Optimizations</i> | | | | | |
| Code factoring [54] | n-to-1 | yes | yes | 0.45 | 0.50 |
| Function inlining [110] | 1-to-n | yes | yes | 0.05 | 0.15 |
| Tail duplication [110] | 1-to-n | yes | yes | 0.05 | 0.15 |
| <i>Obfuscation Techniques</i> | | | | | |
| Two-way opaque predication [42] | 1-to-2 | yes | no | 0.05 | 0.15 |
| Control flow flattening [150] | 1-to-1 | yes | no | 0.05 | 0.15 |
| Branch function redirection [98] | 1-to-1 | yes | yes | 0.05 | 0.15 |
| Opaque predication [42] | 1-to-1 | yes | no | 0.05 | 0.15 |
| <i>Compiler Code Generation Techniques</i> | | | | | |
| Instruction selection [110] | 1-to-1 | no | no | on | on |
| Instruction scheduling [110] | 1-to-1 | no | no | on | on |

tions. First, the compiler optimization techniques we use all have in common that they duplicate code or remove code duplicates, as indicated by their mapping column in Table 2.1. For example, code factoring replaces n identical copies of a code fragment in a program by a single copy. Such 1-to- n or n -to-1 transformations have the effect that a single location in one program version may correspond to two or more locations in another version. Not all tools keep track of such n -to- n mappings, which will result in less accurate diffs. Binary differs that want to give the attacker as accurate information as possible have to handle such cases correctly. This increases their search space.

Furthermore, these transformations can mutate the program’s global CG, and the CFGs of functions. Which transformations mutate the CG and the CFGs are also shown in Table 2.1. Semantically identical functions having different CFGs will make it harder for diffing tools to correctly identify functions and basic blocks that are identical, thus leaving the attacker more code to study manually.

Finally, some of these transformations also thwart CFG reconstruction by hiding direct control flow transfers behind indirect ones, and by changing the code layout. When diffing tools have to rely on poorly reconstructed CFGs, the diffing tools’s ability to map functions will be impaired. Thus, the amount of mapped code fragments will be lower, again resulting in more code fragments the attacker has to study manually.

2.5 Evaluation

In this section, we use our models of real-world attackers to prove the effectiveness of software diversity against patch-based attacks. We will first evaluate these models on undiversified binaries, after which we will compare these results with attacks on diversified binaries. Thus, we will prove the effectiveness of using diversification against attacks that use binary patches to identify vulnerabilities in code. We begin by presenting the case studies we evaluated our attack model on. We then evaluate the different tools and prioritization strategies.

2.5.1 Case studies

In order to evaluate our approach, we have selected four different security patches. The patches we selected all have a different impact on the binary. The patches themselves range from mere changes in constant values to changing the implementation of an algorithm.

Additional validation code The first patch on which we evaluated our approach, hereafter called `bzip2`, fixed vulnerability CVE-2010-0405⁵ in the program `bzip2` by inserting a validation check on an intermediate value as indicated in Figure 2.14(a). In the binary, this corresponds to a short 3-instruction sequence being inserted as shown in Figure 2.15(a).

Off-by-one constant patch Our second patch is an off-by-one fix for vulnerability CVE-2008-3964⁶ in the `pngtest` utility. The fix decrements a hard-coded value of 30 to 29, as shown in Figure 2.14(b). In this case, the off-by-one error is solved by changing a constant value in the source code. Other fixes for off-by-one errors could be by introducing additional validation checks or by changing instructions, and as such are similar to the previous case study. In this case study, we focus on the case in which at the source level, only changed constants constitute the security patch. In particular, we study two instances of fixes for the same off-by-one error.

⁵<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0405>

⁶<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-3964>

```

es = -1;
N = 1;
do {
    if (N >= 2*1024*1024) RETURN(BZ_DATA_ERROR);
    if (nextSym == BZ_RUNA) es = es + (0+1) * N; else
    if (nextSym == BZ_RUNB) es = es + (1+1) * N;
    N = N * 2;
    GET_MTF_VAL(BZ_X_MTF_3, BZ_X_MTF_4, nextSym);
} while (nextSym == BZ_RUNA || nextSym == BZ_RUNB);

```

(a) bzip2 patch

```

#define PNG_tIME_STRING_LENGTH 3029

png_strncpy(time_string,
            png_convert_to_rfc1123(read_ptr, mod_time),
            PNG_tIME_STRING_LENGTH);
time_string[PNG_tIME_STRING_LENGTH] = '\0';

```

(b) png_debian patch

```

#define PNG_tIME_STRING_LENGTH 3029

png_strncpypng_memcpy(time_string,
                    png_convert_to_rfc1123(read_ptr,
                    mod_time),
                    PNG_tIME_STRING_LENGTH);
time_string[PNG_tIME_STRING_LENGTH] = '\0';

```

(c) png_beta patch

Figure 2.14: Source code changes for three of the four patches


```

0x080538b4: movl $0xffffffff,0x60(%esp)
0x080538bc: movl $0x1,0x5c(%esp)
0x080538c4: cmpl $0x1ffff,0x5c(%esp)
0x080538cc: mov $0xffffffffc,%esi
0x080538d1: jb 0x8052892 <BZ2_decompress+386>

```

(a) bzip2 patch

```

0x0804924e: mov 0x124(%esp),%eax
0x08049255: mov %eax,(%esp)
0x08049258: call 0x804a8c0 <png_convert_to_rfc1123>
0x0804925d: mov %eax,0x4(%esp)
0x08049261: movl $0x1e0x1d,0x8(%esp)
0x08049269: movl $0x80d9010,(%esp)
0x08049270: call 0x80806b0 <strncpy>
0x08049275: incl 0x80d900c
0x0804927b: movb $0x0,0x80d902e0x80d902d
0x08049282: jmp 0x8048b44

```

(b) png_debian patch

```

0x804927e: mov 0x124(%esp),%eax
0x8049285: mov %eax,(%esp)
0x8049288: call 0x804a920 <png_convert_to_rfc1123>
0x804928d: mov %eax,0x4(%esp)
0x8049281: movl $0x1e,0x8(%esp)
0x8049289: movl $0x80d9010,(%esp)
0x8049290: call 0x80806b0 <strncpy>
0x804928d: mov (%eax),%ebp
0x804928f: mov %ebp,0x80d9010
0x8049295: mov 0x4(%eax),%edi
0x8049298: mov %edi,0x80d9014
0x804929e: mov 0x8(%eax),%esi
0x80492a1: mov %esi,0x80d9018
0x80492a7: mov 0xc(%eax),%ebx
0x80492aa: mov %ebx,0x80d901c
0x80492b0: mov 0x10(%eax),%ecx
0x80492b3: mov %ecx,0x80d9020
0x80492b9: mov 0x14(%eax),%ebp
0x80492bc: mov %ebp,0x80d9024
0x80492c2: mov 0x18(%eax),%edi
0x80492c5: mov %edi,0x80d9028
0x80492cb: movzwl 0x1c(%eax),%eax
0x80492cf: incl 0x80d900c
0x80492d5: mov %ax,0x80d902c
0x80492db: movb $0x0,0x80d902e0x80d902d
0x80492e2: jmp 0x8048b44

```

(c) png_beta patch

Figure 2.15: Source-code induced mutations in three of the four binary code patches

This fix was part of a larger update from `libpng 1.2.23-beta01` to `beta02`. In that larger update, which also contains patches not related to CVE-2008-3964, the code fragments using the changed constant were patched as shown in Figure 2.14(c). In addition to the changed constant, the call to `png_strncpy` is replaced by a call to `png_memcpy`. The compiler inlines that call and unrolls the loop responsible for copying the actual data in it, so in the patched binary the call to `png_strncpy`, including the preparation of arguments, is replaced by a sequence of `mov` instructions as shown in Figure 2.15(c). The constant value 29 does therefore not occur in the patched binary. We refer to this patch as `png_beta`.

The Debian GNU/Linux distribution made a custom patch for this security vulnerability instead of updating the version of `libpng` in its entirety. The changes relevant to the security issue were extracted and distributed as a separate patch. We made a similar patch for `libpng 1.2.23-beta01`, which contains just the changes in constants. We refer to this patch as `png_debian`. In the binary this patch resulted in four immediate instruction operands being replaced: in two similar fragments a constant operand 30 is replaced by 29 and the absolute address of `time_string[30]` is replaced by that of `time_string[29]`. One of those changed fragments is shown in Figure 2.15(b).

Changed algorithm Finally, we choose the SPEC benchmark program `soplex` as the target of a patch that replaces two (out of several more) calls to `quicksort` with calls to a newly added set of `mergesort` functions. This patch is called `soplex`.

We compiled and statically linked the original and patched source code on Linux with the `gcc 3.2.2` compiler at optimization level `-O3 -fomit-frame-pointer`. Table 2.2 shows the patched binary sizes as well as the sizes of the binary patches generated with the `bsdiff` tool that are typically distributed to the end-users. The three relatively large patch sizes indicate that those patches indeed involve many TIMs as discussed in Section 2.3. The attacker’s goal is therefore to weed those TIMs out by means of `BinDiff`.

In the experiments we performed, we used the SPEC training inputs to collect profile information for `bzip2` and `soplex`. Whenever we report performance overhead, we used reference inputs. For the `png_beta` and `png_debian` benchmarks, we used a PNG image that accompanies the `pngtest` program as input.

Table 2.2: Binary size characteristics of the case studies

| Use case | Binary code size | Binary patch size |
|------------|------------------|-------------------|
| bzip2 | 402 KiB | 4.3KiB |
| png_debian | 503 KiB | 191B |
| png_beta | 503 KiB | 477KiB |
| soplex | 835 KiB | 826KiB |

2.5.2 Representing the results

Evaluating our models results in both recall rates and pruning rates. We represent these results with bar plots, rather than scatter plots so that we can more easily highlight certain combinations of heuristics that would otherwise lie too closely together. An example of such a bar plot is shown in Figure 2.16. In this case, it shows the results obtained with `bsdifff`. The top four bars show the raw results returned by `bsdifff`, which we also call *Set 1* in the figure. This represents the attack model μ_{bsdifff} . The bottom four bars show the model where the attacker has applied heuristics 1 and 2 from Table 2.3 to of Set 1. This represents the attack model $\mu_{\text{bsdifff} \wedge \text{prune changed immediates} \wedge \text{executed}}$. The small vertical lines represent the recall on the top scale. The bars represent the pruning factors on the bottom scale. Because some experiments have a pruning rate of 100%, we modified the logarithmic scale at the bottom to include 100% as its maximum value. Some models do not identify any relevant instructions. For those useless result, we use grey bars for the pruning factor. In some plots, we added circled symbols to refer to from the text.

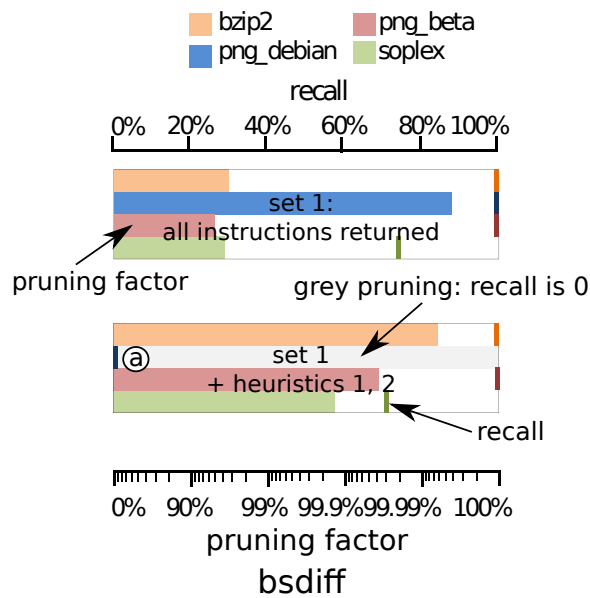
2.5.3 Effectiveness of attacks on undiversified binaries

We start by evaluating how the different tools and heuristics behave on undiversified binaries. The heuristics that we combined in the different models are summarized in Table 2.3.

Figure 2.17 presents the results obtained with the generic binary patching tools `bsdifff` and `xdelta`. The raw output of these tools, which are represented as Set 1 in the figure, are all instructions that these tools did not copy directly from the original binary. Without any additional heuristics, the attacker will have to study all these instructions. In the case of `bzip2`, this leads to a pruning of 96.78% of the instructions in

Table 2.3: The heuristics used for pruning the results of the attack tools

| Number | Heuristic |
|--------|---|
| 1 | Prune instructions with only changes to static operand values |
| 2 | Keep only code covered by typical inputs |
| 3 | Keep only code analyzed by IDA Pro |
| 4 | Extend the disassembly process of IDA Pro |
| 5 | Prune certain patterns in the CFG |
| 6 | Expand the observed instruction window |

**Figure 2.16:** Detailed example of pruning and recall rates

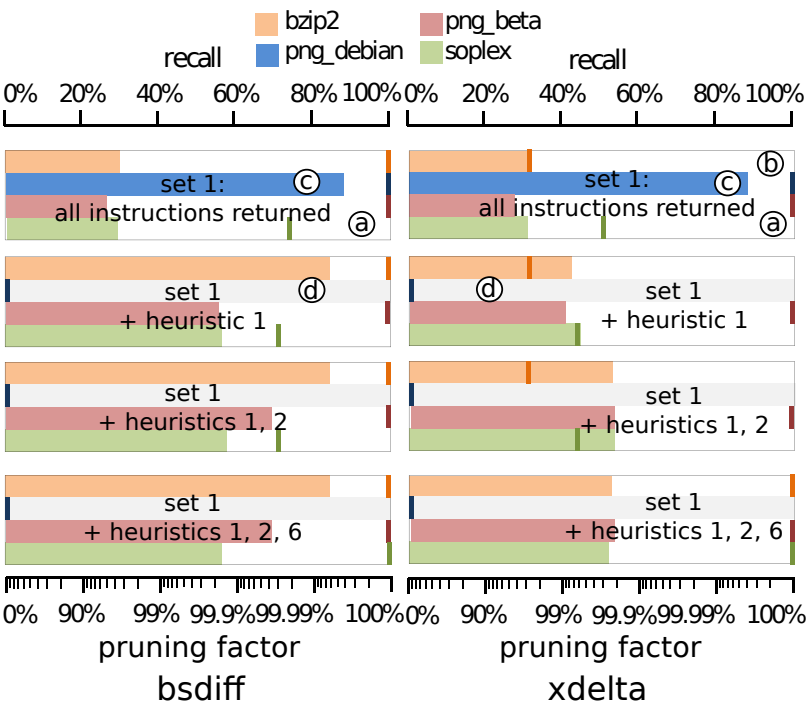


Figure 2.17: Pruning rates (bars) and recall rates (lines) for bsdiff and xdelta.

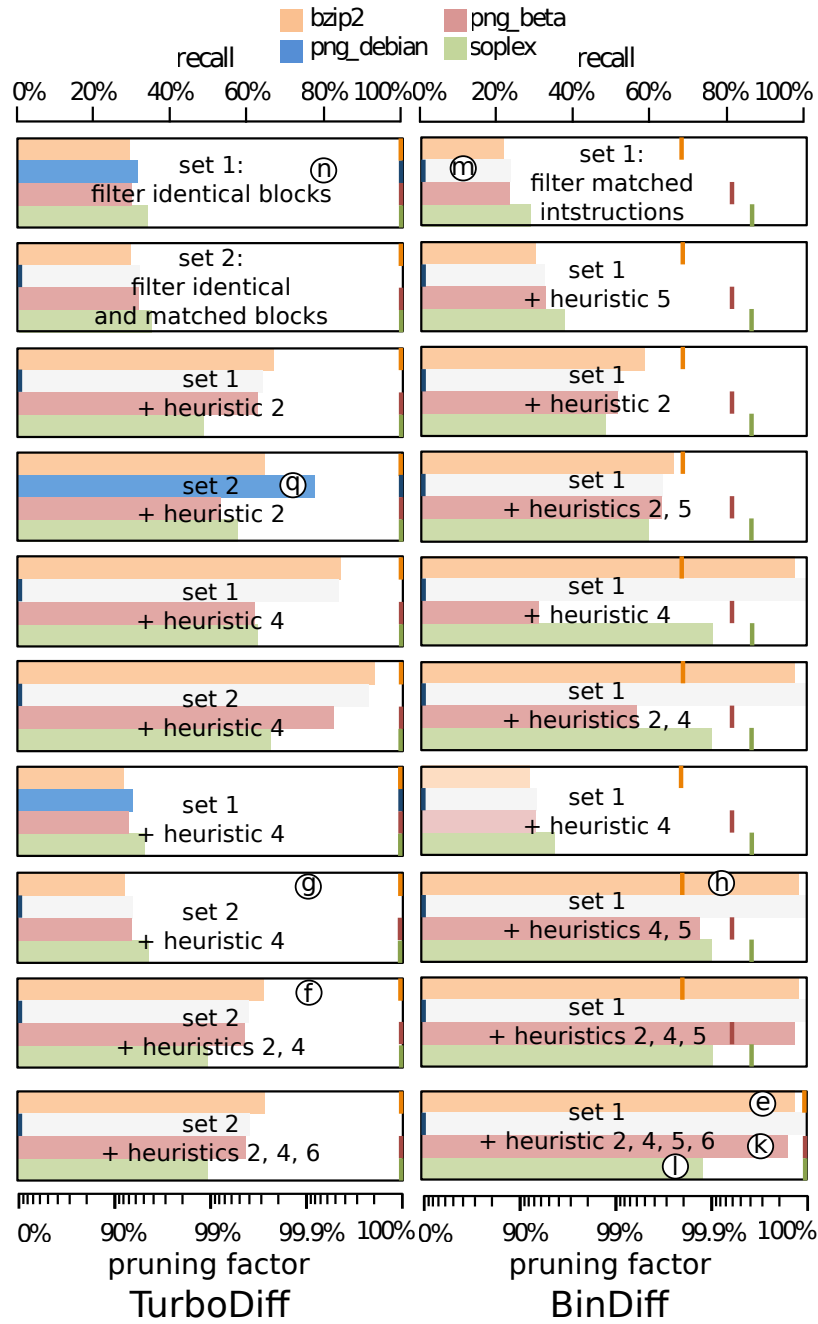


Figure 2.18: Pruning rates (bars) and recall rates (lines) for patchdiff2 and BinaryDiffer.

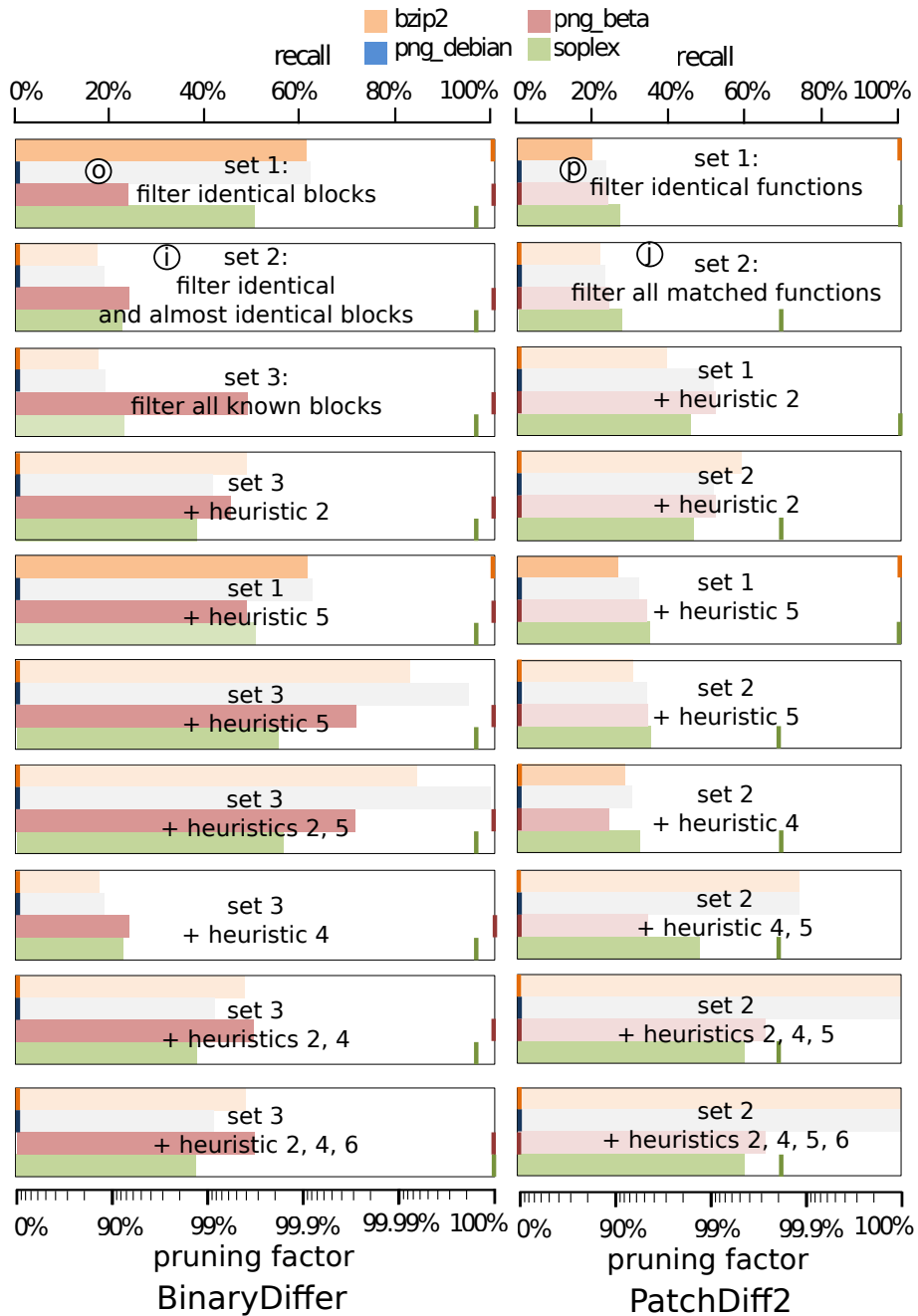


Figure 2.19: Pruning rates (bars) and recall rates (lines) for patchdiff2 and BinDiff.

the case of `bsdifff`, and 97.25% in the case of `xdelta`. This means that, even though just 3 instructions were added by the patch, about 3% of the binary is reported as changed. Although the amount of changes is overestimated by these tools, they do not report all SCIMs. This happens because some instruction sequences such as function entry and return sequences can be copied verbatim from the unpatched binary, even though they are part of a SCIM. This can be seen in (a) for `soplex` and in (b) for `bzip2`.

In the case of `png_debian`, the raw tool results are already optimal. In this case, `bsdifff` and `xdelta` encode precisely the SCIMs to the four immediate operands. The patch thus points the attackers directly to nothing less than the relevant instructions (see (c)).

Adding the different heuristics described in Section 2.3 that are applicable to the generic binary patching tools, we see a significant increase in the pruning rates. However, these heuristics can also prune too many instructions. The attack models $\mu_{\pi \wedge \text{prune changed immediates}}$, denoted by (d), no longer detect patched code where the SCIM is contained in changes to operands. We observe that this is the case for the minimal patch of `png_debian`.

Figures 2.18 and 2.19 present the results obtained with ten combinations of heuristics for the specialized diffing tools TurboDiff, BinDiff, Patchdiff2 and BinaryDiffer.

Which tools and heuristics perform best depends on the use case. For `bzip2`, we see in (e) that BinDiff gives the best pruning, but only with the attack model $\mu_{\text{BinDiff} \wedge \text{exec} \wedge \text{only disass} \wedge \text{prune patterns} \wedge \text{expand window}}$. Note that the expanded window makes the recall 100%. As we observe in (f), TurboDiff prunes less, but contrary to BinDiff, TurboDiff identifies all relevant instructions without the need to expand the instruction window. When only considering direct identification of relevant instructions, TurboDiff in general gives a higher recall than BinDiff. Compare, for instance, (g) to (h). Furthermore, we can see in (i) that BinaryDiffer reports that the changed code in `bzip2` as very similar. Similarly, observe in (j) that patchdiff2 labels this code fragment as similar.

For `png_beta`, we note in (k) that BinDiff scores best. As shown in (l), it also works best for `soplex`. For `png_debian`, we see in (m) that BinDiff's abstractions cannot identify any relevant instruction. TurboDiff identifies the changed immediate operands (see (n)), but only when using the appropriate heuristics. When those heuristics are ap-

plied for any other patch, they do not at all achieve the highest pruning rate. While BinaryDiffer and patchdiff2 have higher pruning rates for `png_debian` than the BinDiff or TurboDiff (see (o) and (p)), neither is actually able to find the SCIM.

This means that for the minimal patch of `png_debian`, an attacker could better use a binary patch tool, rather than one of the IDA-Pro based diffing tools. However, this “shortcut” of using a byte-level diffing tool is available only because all syntactic changes in this use case are SCIMs. When such a minimal security fix involving only changed constants is combined with other (non-related) fixes as in `png_debian`, the patch includes many more TIMs, which prevent this method from being used as a shortcut.

The highest pruning factors obtained with BinDiff are 99.988% (e), 99.986% (k) and 99.909% (l). As the fractions of irrelevant instructions in those cases are 99.997%, 99.986%, and 99.923% respectively, BinDiff proves to be able to prune more than 99.98% of all irrelevant instructions for those three use cases.

The worst match rates are those of patchdiff2, which only finds changes to `soplex` and `bzip2`. The best match rates are those of TurboDiff, which finds all SCIMs for some combinations of heuristics, such as in (q). However, as already remarked, BinDiff’s maximum pruning factors are higher in all cases, except for the mutations in `png_debian` which it fails to detect.

This demonstrates that for some types of patches and undiversified code, diffing tools and heuristics are indeed valuable attacker tools. For other types of patches however, they are much less effective than the generic binary patch tools. Moreover, as an attacker typically does not know beforehand which types of patches have been applied, he will be hindered by not being able to fine-tune his heuristics.

2.5.4 Diversification

Now that we have shown that attackers, as represented by our attack models, can indeed correctly identify SCIMs in binary code, we can investigate if introducing software diversity can thwart these attacks. Furthermore, we compare the overhead that is introduced by diversity to the effect it has on an attacker.

We present and discuss results for binaries diversified at two levels of diversification. Experiments with other settings confirm the trends

presented here. For the four use cases and two diversification levels, we generated 10 pairs of an unpatched and a patched binary, using 80 different PRNG seeds. We then applied 60 combinations of tools and heuristics on the 8×10 pairs of patched and unpatched program versions, for a total of 4800 diffing attempts.

In Figure 2.20, the results of these 4800 attempts of all of our evaluated diffing tools on different diversity settings are plotted next to the Pareto-optimal results (as black symbols) from Figures 2.17, 2.18, and 2.19.

In a first experiment, the only diversification used is code layout randomization. This is shown by the small, red dots in the figure. We see that this layout randomization has an effect on all tools. The tools that are affected most are the binary patch tools, for which in some experiments less than 80% of the instructions can be pruned, and for which all experiments perform worse than on the undiversified benchmarks. The IDA Pro-based tools are affected as well, albeit less. This is due to a lesser accuracy of the CFGs constructed by IDA Pro. This may come as some surprise, but this is due to IDA Pro assuming that the instruction immediately after a call is returned to after the call, and thus belongs to the same function's CFG. However, this is not the case in the relayed code. Call instructions that never return end an instruction chain. The subsequent instruction therefore need no longer belong to the same function. Therefore, IDA Pro incorrectly groups code from different functions, resulting in lower match rates of the IDA Pro-based diffing tools, even when only little diversity is introduced.

In the next experiments, we increase the diversity applied to the binaries by also applying other diversity transformations provided by Proteus. We evaluated and compared two different diversification settings: low diversity (represented by the medium-sized, blue circles in the figures) and high diversity (represented by the large, green circles in the figures). These settings correspond to the settings also used by Anckaert [7]: the low diversity setting is the p_1 default set of Proteus parameters, and the high diversity setting is the p_3 set of parameters. The exact parameters for Proteus for each of these settings are summarized in Table 2.1.

A first observation of the results is that, with the right heuristics, the tools can still retrieve most relevant instructions from the diversified binaries. Which heuristics are the right ones differs from experiment to experiment. However, the tools and heuristics only retrieve these

relevant instructions at pruning rates less than 90%. This means that the amount of code the attacker has to examine has grown 100-fold. A second observation is that the binaries transformed at the higher level of diversification indeed have a smaller pruning rate than when the low diversity setting is used, although the difference is rather small in most cases. Third, in all tool results, there is a clear grouping of pruning rates greater than 80%, and pruning rates that are significantly lower. This distinction is caused by the executed-code-only heuristic that prunes all unexecuted code. Since only about 20% of the code is executed, the attack models that only use executed code have a higher pruning rate. This shows that even the most simple use of dynamic information can reduce the attack effort.

Finally, we note that occasionally BinDiff took more than 24 hours to produce the diffing result when being executed on an Intel Core i7 running at 3.4GHz, or consumed all available memory and crashed. This is probably caused by BinDiff splitting the code into too many functions to analyze and compare. This is detrimental to the attacker, who can waste time on waiting results that never emerge.

Table 2.21 shows that diversification as applied by Proteus comes with considerable overhead in file size. While the binaries already grow by 20% with low diversity, the patches that would be distributed typically become between 1 and 2 orders of magnitude larger. For the small `png.debian`, the patch becomes 1690 times bigger. Similarly, Figure 2.22 shows that stochastic diversification results in considerable slowdowns. Furthermore, we see that the slowdown does not correlate well with the achieved pruning factors.

Whether or not the overheads in patch size, binary size, and execution time are acceptable is a decision that must be made by the software vendor. This decision will depend on multiple factors, such as the severity of the fixed vulnerability, its ease of exploitation, and the speed with which users apply software updates.

2.6 Conclusion

We introduced an abstract attacker model for patch-based attacks. We described how different real-world attack tools can be used by attackers, and we described different heuristics attackers can use to improve the results of these tools. We instantiated the abstract attacker model with combinations of attack tools and diffing tools.

The different attack models allowed us to show that the different attack tools and strategies indeed allow attackers to efficiently recover source-code induced mutations when the patched binaries are not protected against patch-based attacks. We then used our models to show that software can indeed be successfully protected against patch-based attacks by applying diversification.

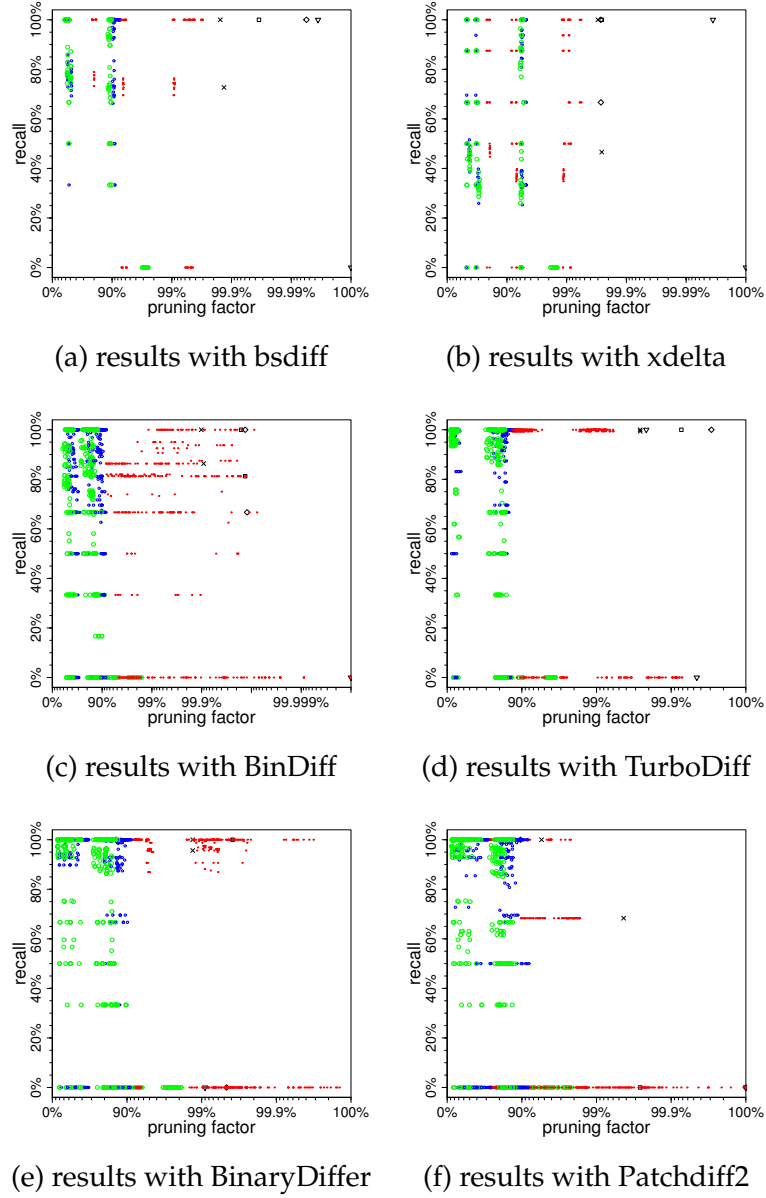


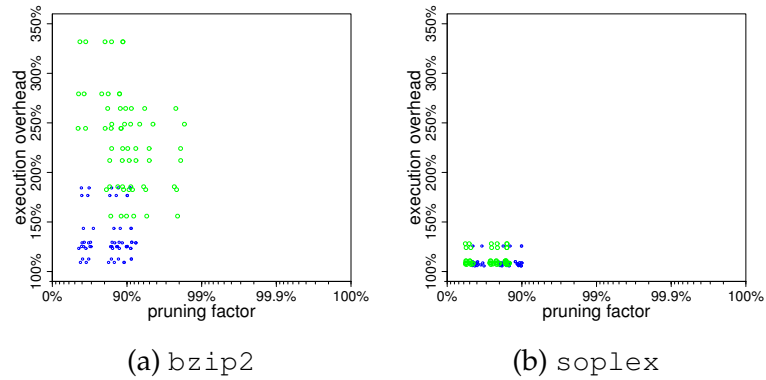
Figure 2.20: Comparison of all tools on diversified and undiversified binaries. The black symbols represent the Pareto-optimal points of the different benchmarks without diversity. The small, red dots represent the experiments with layout randomization, the medium-sized, blue circles represent experiments that are diversified with low diversity settings, and the large, green circles represent experiments that are diversified with high diversity settings.

| Benchmark | avg. patch size increase | avg. binary size increase |
|------------|--------------------------|---------------------------|
| bzip2 | 4631% | 20% |
| png_beta | 168946% | 20% |
| png_debian | 2743% | 20% |
| soplex | 4011% | 20% |

(a) Light diversification

| Benchmark | avg. patch size increase | avg. binary size increase |
|------------|--------------------------|---------------------------|
| bzip2 | 5084% | 28% |
| png_beta | 184772% | 29% |
| png_debian | 3040% | 29% |
| soplex | 4482% | 31% |

(b) Heavy diversification

Figure 2.21: Size overhead of diversification**Figure 2.22:** The overhead in execution time is compared with the pruning factors of BinDiff. The medium-sized, blue circles represent experiments that are diversified with low diversity settings, and the large, red circles represent experiments that are diversified with high diversity settings.

Chapter 3

Iterative feedback-driven diversification

3.1 Introduction

In Chapter 2 we have shown how existing diversifying transformations are effective against patch-based collusion attacks. However, those transformations as applied by Proteus introduce significant overhead. In this chapter, we will extend the Proteus diversification framework to mitigate these problems. We call our extended framework Glaucus, after the Greek mythological deity. Like Proteus, Glaucus is a sea-god. He came to the help of sailors in storms, because he himself once had been a mortal fisherman, or (according to other writers), had been thrown overboard the ship Argo and been rescued by Zeus¹. In a way, this is similar to what our framework does. It is an iterative framework that uses knowledge of previous iterations to guide the transformation process in the current iteration, just like Glaucus used his prior sea-faring experience of being in a storm to help sailors in need.

Existing program diversification frameworks apply transformations randomly throughout the program binary [7, 97]. However, not all these transformations are necessary to thwart an attacker. If the diffing tools cannot successfully match some functions after applying a single transformation to them, there is little reason to apply more transformations to these functions, and introduce unnecessary overhead.

¹[...]; and the sailors, safe in port, shall pay their vows on the shore to Glaucus, and to Panopea, and to Melicerta, Ino's son. – Publius Vergilius Maro, *Georgica* I, 432, translation by H. R. Fairclough. For more information, see for example the *Harpers Dictionary of Classical Antiquities* (1898) by Harry Thurston Peck.

In this chapter, we modify the existing diversifying framework Proteus to mitigate this problem. We iteratively use feedback from the attack tools to guide the diversification process. After every iteration, Glaucus evaluates an attack model to compare the patched binary that was generated in that iteration to the unpatched binary. The result of this comparison is then used as input to the next iteration, in which Glaucus then applies the same transformations as the previous iteration to functions that are not matched. Functions that are matched by the attack tool will be transformed differently and more heavily. Furthermore, BinDiff tells the attacker the selector it used to match a pair of functions. We use this information to select which specific transformations to apply the next iteration.

In this chapter, we first describe in detail how our feedback-guided iteratively diversifying compiler works (Section 3.2). We continue by evaluating the effectiveness and efficiency of this diversifying compiler (Section 3.3). This is followed by a more thorough discussion of the results (Section 3.4).

3.2 Feedback-guided iterative diversification

Figure 3.1 shows the tool flow of Glaucus. Compared to the tool flow of Proteus in Figure 2.9, we have replaced the original diversification backend with a framework that can be applied iteratively based on feedback obtained from one or more attack models we want to defend against.

In our proof-of-concept implementation, Glaucus consists of the standard GCC 4.6 compiler plus an evolution of Proteus [7]. As opposed to the original design of Proteus, the application of the transformations is no longer a purely stochastic process. Instead, we use a configurable decision logic. This decision logic takes into account profile information, feedback obtained from the diffing tools, and a log of the transformations applied in the previous iteration. A PRNG is then used to select among transformations chosen by the decision logic.

3.2.1 Attack model

With Glaucus we focus on using the diffing results obtained from BinDiff. In Chapter 2 we already demonstrated that on most use cases, the combination of BinDiff and appropriate heuristics outperforms the

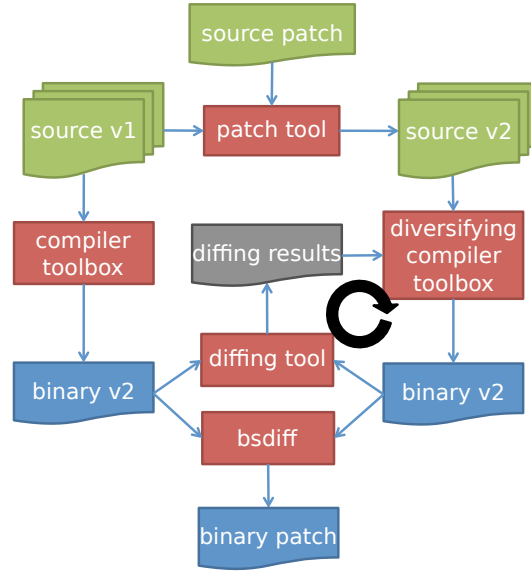


Figure 3.1: Iterative tool flow of Glaucus for generating protected patches.

other diffing tools. Furthermore, BinDiff internally uses up to 19 heuristics to match functions, and reports which heuristic was used to match each pair of functions. We use this information as feedback for iterative diversification. We focus on thwarting BinDiff’s output without any pruning heuristics applied. Pruning heuristics use this output as a starting point; by making the starting point useless we aim to make the improvements upon it useless as well.

3.2.2 Diversifying transformations

We selected five transformations to be used by our diversifier. They are similar to those included in Proteus. So here, our description focuses on the differences with their implementation in Proteus, on the reasons for their inclusion in our iterative framework, on their impact on performance, and on code size.

Code layout randomization The iterative code diversifier randomizes the order in which basic block chains are placed in the executable’s code section. A basic block chain is a sequence of BBLs that have to be placed consecutively to each other in the generated binary. They are

linked together through their fall-through path, and end with a basic block that performs a control transfer without fall-through path. These BBLs have to be placed sequentially²; if not control flow through their fall-through path would no longer be correct.

As a result, function bodies are not necessarily stored contiguously. This complicates IDA Pro's partitioning of the disassembled code in functions and its construction of the CG and the CFGs.

The performance impact of reordering basic block chains is minimal. The impact on code size incurred by this transformation is small. For example, in the x86 ISA, branch targets are encoded as the offset between the address of the next instruction and the branch target [81]. Since these offsets are small in undiversified programs, they can be encoded efficiently. However, when randomizing the locations, the average offsets become much larger, and require more space to be encoded. In the case of 32 bit x86 binaries, the overhead introduced by this transformation on most branch instructions is 3 bytes. On our benchmark binaries, this results in 4 to 5% total code size increase.

Conditional branch flipping To thwart some very simple matching heuristics, simple CFG transformations suffice. Conditional branch flipping inverts the condition on the instruction. For example, a branch-if-greater instruction is transformed into a branch-if-less-or-equal instruction. To compensate for this inversion, its successor BBLs in the CFG are swapped, with unconditional jumps added if necessary.

Binary differs that consider distinct opcodes and jump conditions as a reason not to match basic blocks and functions are thwarted with this technique. In particular, BinDiff's *Instruction Signature* selector matches functions based on which instructions occur in the functions [159]. Because inverting the condition of a branch changes which instructions are in a function, branch flipping can thwart this matching strategy.

This transformation has almost no influence on performance. The influence on code size is minimal as well.

Partial control flow flattening As discussed in Chapter 2, control flow flattening transforms a function's CFG into one where all original basic blocks have the same predecessor and successor [149].

Our iterative diversification tool can flatten parts of functions;

²Unconditional jumps could be inserted in the middle of a chain to break it.

hence the name partial flattening. The reason is that flattening in our context does not aim for obfuscation, but for thwarting diffing tool heuristics based on CFG topology properties. Changing parts of the CFGs, such as the coldest parts, suffices for that purpose.

Since function flattening introduces code to redirect the control flow, it can introduce a significant performance overhead, that can be limited by applying flattening to cold code only, its impact on performance. Similarly, the code size overhead scales with the number of basic blocks. For every function to which it is applied, it adds 4 instructions for the control flow redirection, and additionally 7 instructions per flattened basic block. The relatively large overhead per BBL is due to the spilling of registers on the stack that are modified by the control flow redirection code.

Two-way opaque predicate insertion As already explained in Chapter 2, two-way opaque predicate insertion involves the duplication of (part of) a basic block, and the insertion of a random branch condition [42]. Its implementation is identical to the one in Proteus.

The duplicate blocks can be transformed independently by later transformations. Two-way opaque predicate insertion targets very simple matching heuristics that use static instruction counts and information about control flow. BinDiff has multiple such heuristics, such as *Edges Flowgraph* and *Instruction Count*.

Its impact on performance can be limited by applying the transformation to cold code only. Its impact on code size depends on the size of the duplicated code.

Branch function insertion and call function insertion As explained in Chapter 2, branch functions are functions that do not return to their caller; instead they transfer control to a different address computed from the return address and an offset passed to the branch function as a parameter. So a direct jump or fall-through between two basic blocks can be replaced by a call and an indirect jump based on the call's arguments. Figure 3.2(a) shows an unconditional jump, which is transformed into a call to the branch function in Figure 3.2(b). We refer to this as *branch function insertion*.

Direct control transfers now use dynamically computed values. This makes correctly partitioning the BBLs into functions harder. When control transfer targets can be statically determined, IDA Pro can follow

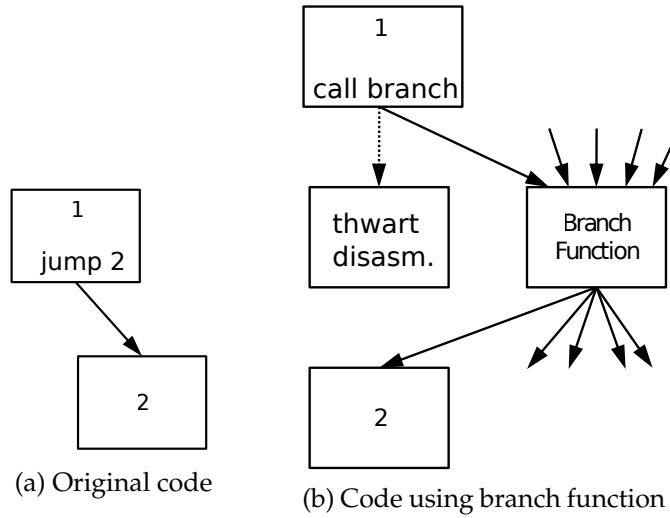


Figure 3.2: Branch function insertion

the control flow of a function, which is no longer the case after applying this transformation. Furthermore, IDA Pro assumes that a function call returns at the instruction following the call. However, with our branch functions, the control flow continues at the target of the direct control transfer, rather than at the next instruction. This is then exploited during code layout randomization, where the code following a call to the branch function can be code from any basic block *in the entire program*. This will help in making IDA Pro incorrectly partition the basic blocks into functions.

In addition to *Branch Function Insertion*, we also apply *Call Function Insertion*. Such call functions are used to replace direct function calls by indirect control flow. It suffices to emulate a call by pushing its return address and call target on the stack and to transfer control to the branch function. Branch functions can therefore thwart both CFG-based and CG-based matching heuristics.

Branch Function Insertion can be applied in different places to thwart different selectors used by BinDiff. By inserting a branch function in the entry block of a function, most of its body becomes disconnected from its entry point. By inserting a branch function before a call, that call is not removed, but it becomes disconnected from the preceding code. This breaks selectors based on the call graph.³

³We also experimented with adding an additional branch function call at the func-

Branch functions clearly introduce a performance penalty when they are inserted in frequently executed code. Each jump is statically replaced with 3 instructions. Dynamically, one of these instructions calls the branch function, which requires 5 additional instructions to execute.

3.2.3 Transformation Selection

The selection of diversifying transformations to be applied is configured on the basis of a set of rules encoded in a rule set table. The table used for the experimental evaluation in Section 3.3 is depicted in Table 3.1.

Each row in the rule set table specifies a *necessary condition* to apply a transformation. A row with signature S (we refer to the BinDiff manual [159] for a detailed description of the exact signatures), relative weight W , iteration I and transformation T specifies that in any iteration $i \geq I$, the transformation T can be applied to basic blocks with a relative weight $w \leq W$ in functions reported by the diffing tool to have been matched on the basis of signature S in iteration $i - 1$.

We take profile information into account for the *weight* of a BBL. This is its execution count given a training profile, multiplied by the number of instructions in the block. The total weight of the program equals the summed weights of all blocks. The relative weight of a block is obtained by dividing its weight by the total program weight. In case a transformation transforms multiple BBLs, the used weight is the sum of all the individual BBL weights.

For example, the first row in Table 3.1 indicates that conditional branches may be flipped in any iteration, in any function matched by means of the “Hash Matching” signature, independent of the weight of the block of the branches. The third row indicates that from the first iteration on, two-way predicates can be inserted in or before basic blocks that have weight zero (i.e., that are not executed) in functions that were matched on the basis of the “Edges Flowgraph” signature. It is clear that gradually, transformations with higher overhead will be considered, and that their application onto ever hotter blocks is considered.

We designed the rules in Table 3.1 as follows. We started by investigating the selectors used by BinDiff. We aggregated the number

tion’s return address; however, this showed no improvement over having the branch function only before the call instruction.

of matched instructions by selector. Thus, we were able to focus on the heuristics that were most commonly used by BinDiff. We investigated which program code characteristics these algorithms depend on. Knowing which program characteristics are affected by the different transformations enabled us to select the right program transformation to thwart the different algorithms. We used our knowledge of the runtime overhead of the transformations to decide the maximum weight of code on which to apply the transformations.

This decision logic is extensible to more transformations and other diffing tools, assuming they return info on used matches. Users of Glaucus wanting to add a new transformation can just add it to the rule set in a similar fashion to how we designed the default Glaucus rule set. Like us, they can include knowledge of the transformation's effect on the diffing tools, and knowledge of the overhead in the modified rule set. To extend this process to other diffing tools, we have to include feedback from these diffing tools into Glaucus. From the diffing tools we evaluated in Chapter 2, we can easily extract the list of functions that have (not) been matched. These can already be used as input to Glaucus. However, BinDiff is the only tool that shows which matching heuristics were used to match pairs of functions. We could extend existing tools to export this information. Alternatively, we could opt to transform matched functions with a fixed set of transformations, rather than a set of transformations that is based on diffing feedback. Furthermore, Glaucus could compute some of the code characteristics on which the diffing tool bases its heuristics, and use those to guide the feedback process.

We manually tuned the parameters in our decision logic by evaluating the performance impact on `bzip2`. Users that apply Glaucus on their own software can of course tune the parameters to fit their use case best, or they can re-use the default Glaucus parameters.

During the successive iterations, a diversification strategy is built for each function. This strategy is a log of the applied transformations: the program point where each transformation was applied, the iteration in which it was applied, the reason why it was applied (i.e., the BinDiff signature targeted with it, which comes from the rules table), and the PRNG seed that was used for selecting that transformation. An example table for a function `f()` is shown in Table 3.2.

Initially, i.e., after iteration zero in which only the code layout is randomized, each function's strategy is empty. Then during each it-

Table 3.1: Glaucus default rule set

| Signature | Weight | Iteration | Transformation |
|--------------------------|---------|-----------|---|
| Hash Matching | 1.00000 | 1 | Conditional Branch Flipping |
| Edges Flowgraph | 1.00000 | 1 | Conditional Branch Flipping |
| Edges Flowgraph | 0.00000 | 1 | Two-way Opaque Predicate Insertion |
| Edges Flowgraph | 0.00005 | 5 | Two-way Opaque Predicate Insertion |
| Edges Flowgraph | 0.00005 | 6 | Partial Control Flow Flattening |
| Edges Callgraph | 0.00000 | 1 | Branch Function Insertion (anywhere) |
| Edges Callgraph | 0.00005 | 5 | Branch Function Insertion (anywhere) |
| Edges Callgraph | 0.00000 | 1 | Two-way Opaque Predicate Insertion |
| Edges Callgraph | 0.00005 | 5 | Two-way Opaque Predicate Insertion |
| Instruction Signature | 1.00000 | 1 | Conditional Branch Flipping |
| Call Sequence | 0.00000 | 5 | Call Function Insertion |
| Call Sequence | 0.00005 | 7 | Call Function Insertion |
| Call Sequence | 0.00015 | 9 | Call Function Insertion |
| Call Sequence | 0.00000 | 11 | Branch Function Insertion (before calls) |
| Call Sequence | 0.00005 | 13 | Branch Function Insertion (before calls) |
| Call Reference | 0.00000 | 5 | Call Function Insertion |
| Call Reference | 0.00005 | 7 | Call Function Insertion |
| Call Reference | 0.00015 | 9 | Call Function Insertion |
| Call Reference | 0.00000 | 11 | Branch Function Insertion (before calls) |
| Call Reference | 0.00005 | 13 | Branch Function Insertion (before calls) |
| Call Sequence (Exact) | 0.00000 | 7 | Call Function Insertion |
| Call Sequence (Exact) | 0.00005 | 9 | Call Function Insertion |
| Call Sequence (Exact) | 0.00015 | 11 | Call Function Insertion |
| Call Sequence (Exact) | 0.00000 | 13 | Branch Function Insertion (before calls) |
| Call Sequence (Exact) | 0.00005 | 15 | Branch Function Insertion (before calls) |
| Call Sequence (Topology) | 0.00000 | 7 | Call Function Insertion |
| Call Sequence (Topology) | 0.00005 | 9 | Call Function Insertion |
| Call Sequence (Topology) | 0.00015 | 11 | Call Function Insertion |
| Call Sequence (Topology) | 0.00000 | 13 | Branch Function Insertion (before calls) |
| Call Sequence (Topology) | 0.00005 | 15 | Branch Function Insertion (before calls) |
| Hash Matching | 0.00000 | 2 | Branch Function Insertion (in entry blocks) |
| Edges Flowgraph | 0.00000 | 2 | Branch Function Insertion (in entry blocks) |
| Edges Callgraph | 0.00000 | 2 | Branch Function Insertion (in entry blocks) |
| Instruction Signature | 0.00000 | 2 | Branch Function Insertion (in entry blocks) |
| Call Sequence (Exact) | 0.00000 | 2 | Branch Function Insertion (in entry blocks) |
| Call Sequence (Topology) | 0.00000 | 2 | Branch Function Insertion (in entry blocks) |
| Call Sequence | 0.00000 | 2 | Branch Function Insertion (in entry blocks) |
| Call Reference | 0.00000 | 2 | Branch Function Insertion (in entry blocks) |
| String Reference | 0.00000 | 2 | Branch Function Insertion (in entry blocks) |
| Hash Matching | 0.00005 | 3 | Branch Function Insertion (in entry blocks) |
| Edges Flowgraph | 0.00005 | 3 | Branch Function Insertion (in entry blocks) |
| Edges Callgraph | 0.00005 | 3 | Branch Function Insertion (in entry blocks) |
| Instruction Signature | 0.00005 | 3 | Branch Function Insertion (in entry blocks) |
| Call Sequence (Exact) | 0.00005 | 3 | Branch Function Insertion (in entry blocks) |
| Call Sequence (Topology) | 0.00005 | 3 | Branch Function Insertion (in entry blocks) |
| Call Sequence | 0.00005 | 3 | Branch Function Insertion (in entry blocks) |
| Call Reference | 0.00005 | 3 | Branch Function Insertion (in entry blocks) |
| String Reference | 0.00005 | 3 | Branch Function Insertion (in entry blocks) |

Table 3.2: Diversification strategy for a function $f()$

| Transformation | Program Point | Iteration | Signature | Random Seed |
|-----------------------------|---------------|-----------|---------------|-------------|
| Conditional Branch Flipping | BBL 2356 | 1 | Hash Matching | 14562 |
| Call Function Insertion | BBL 2347 | 3 | Call Sequence | 16382 |
| Call Function Insertion | BBL 2349 | 3 | Call Sequence | 16382 |

eration, three actions can be performed on the strategy: it can remain identical, it can be extended, or it can be adapted. Let us assume we are in iteration $i \geq 1$.

1. When a function is not matched by BinDiff according to the feedback of iteration $i - 1$, the strategy remains untouched. This happens when the strategy succeeded in thwarting BinDiff completely for this function. It implies that we will apply exactly the same transformations in iteration i .
2. When a function is matched by BinDiff with some signature s according to the feedback of iteration $i - 1$, and that s is not the same as the last signature s' in the strategy, this implies that the strategy was successful in thwarting matching based on s' , but not in thwarting matching based on s . In this case, the strategy is extended: from all transformations that meet the necessary conditions as specified by the rule table, one is selected randomly and appended to the strategy. By construction, this will be a transformation that targets signature s .
3. When a function is matched by BinDiff with a signature s that already occurs in the transformation strategy for that function, this implies that the strategy was not successful for this function. We then remove the transformations from the last iteration from the strategy, and replace them with a new selection of transformations. This selection happens on the basis of another random seed. Furthermore, we now select one more transformation than we selected in the previous iteration. Moreover, the set of applicable transformations may have become bigger because new rules become applicable in later iterations.

Given these rules, the diversification strategy from Table 3.2 can be interpreted as follows. Assume, e.g., we are now after iteration 5, “Conditional Branch Flipping” succeeded in thwarting signature

“Hash Matching” in iteration 1. From then on signature “Edges Flow-graph” became the target. Two transformations needed to be applied in thwarting this signature, which was discovered in iteration 3. When all three transformations are applied in combination with code layout randomization, BinDiff is no longer able to match function $f()$.

3.3 Evaluation

We use the use cases from Chapter 2 as benchmarks to evaluate Glaucus’s success of improving the attack cost, and the overhead in code size and execution time.

3.3.1 Diffing results

Using the rule set of Table 3.1, we generated multiple diversified binaries for our four use cases. We used the automated attack model $\mu_{\text{BinDiff} \wedge \text{extend IDA}}$ from Chapter 2 to simulate an attacker using IDA Pro in combination with BinDiff. We use this model after every iteration to provide feedback for Glaucus.

For `bzip2`, the chart in Figure 3.3 (a) depicts the pruning rate the attacker achieved with $\mu_{\text{BinDiff} \wedge \text{extend IDA}}$. The different selectors BinDiff used to match code, i.e., code that the attacker will prune, are indicated by the color of the bars. For the precise meaning of the different matching heuristics, we refer to the BinDiff manual [159]. For each iteration, the total height of the bars indicates the matches reported by BinDiff. Some of those matches, in particular the ones based on low quality heuristics and signatures, are false positive matches, however.

To study the influence of false positive matches, we evaluate if the instructions BinDiff returns as a match are indeed in the same function. We computed a ground truth for function matches for each instruction, $\tau_{\text{function match}}$. It consists of tuples of instructions, where the first instruction is from the original binary, and the second instruction is from the patched binary. The ground truth contains all such pairs for which the functions indeed correspond between the two binaries. Figure 3.3 (b) shows how many of BinDiff’s instruction matches are in this set, i.e., $\hat{\mu}_{\text{BinDiff} \wedge \text{extend IDA}} \cap \tau_{\text{function match}}$, where $\hat{\mu}$ is the *mapping* returned by the attack model, rather than a list of matched instructions. While this fraction of instructions is of course most useful to an attacker, he has no ground truth to compute this fraction himself. We provide these mea-

asures thus as an indication for the defender. The charts in Figure 3.4 display the same fractions, but this time with an indication of the match quality according to the BinDiff manual [159] rather than the heuristics used to match the code. The numbers above or in the different bars indicate how many of the relevant instructions (i.e., the instructions in red in Figure 2.15) are found in each category. Numbers above the bars indicate the number of those instructions that are in the unmatched part of the code, i.e., the part of the code within which BinDiff provides no help to the attacker at all. As in Chapter 2, we see that BinDiff initially incorrectly matches one instruction. This instruction is located in a function that is matched correctly, but with a poor heuristic.

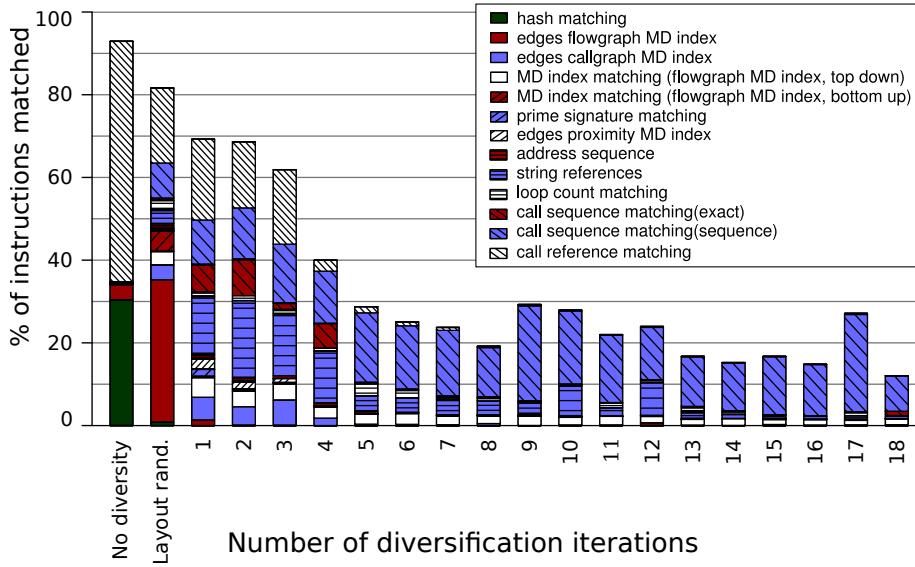
Note that the total number of relevant instructions can vary from one iteration to the other when the relevant instruction sequences are duplicated by transformations.

The leftmost “iteration” in all charts corresponds to diffing the unpatched binary with the undiversified patched binary. It is clear that in that case BinDiff is performing very well. As indicated in green, many function matches are found by computing hashes over the ordered instructions in the functions. The hashes neglect immediate operands and thus ignore changed offsets and changed absolute addresses. Most of the matches are found recursively on the CG through so called “call sequence”-based selectors: these build on the assumption that functions whose callers match, are likely matches themselves.

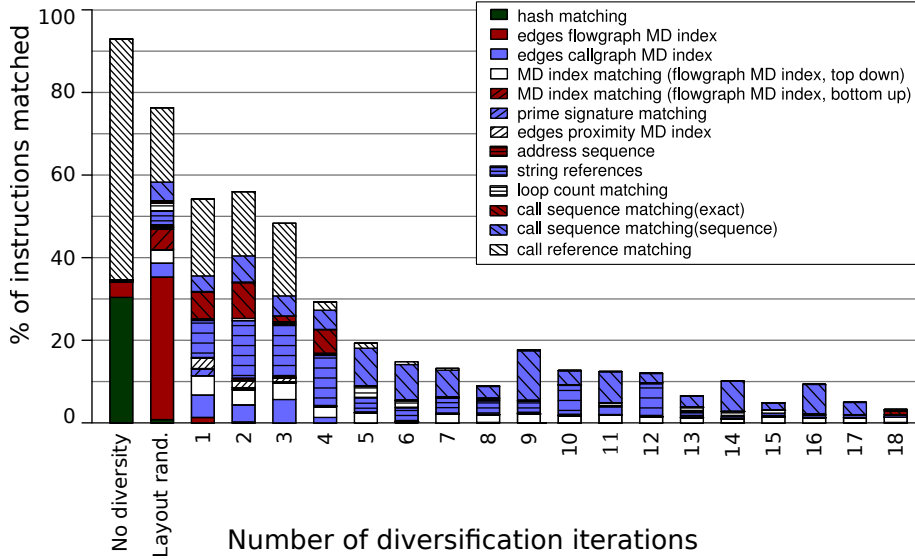
The second iteration from the left in both charts is iteration zero, in which only code layout randomization is applied. It is clear that while BinDiff is hampered by this randomization, it still does a pretty good job. The matching is now mainly based on more abstract CFG properties, however, that rely less on the order and occurrence of individual instructions.

As soon as we start diversifying the code, the effectiveness of BinDiff starts to drop. Almost immediately, the “very good” quality matchers start to fail and the lower quality metrics take over. The matcher based on the strings that are referenced in functions then becomes quite important, along with the different selectors based on the CG. As more and more diversification is introduced during our iterative approach, the amount of matched code drops significantly, and the amount of correctly matched code drops even lower.

In this experiment, the best result is obtained after 14 iterations. For the binary generated in that iteration, BinDiff can match about 15% of

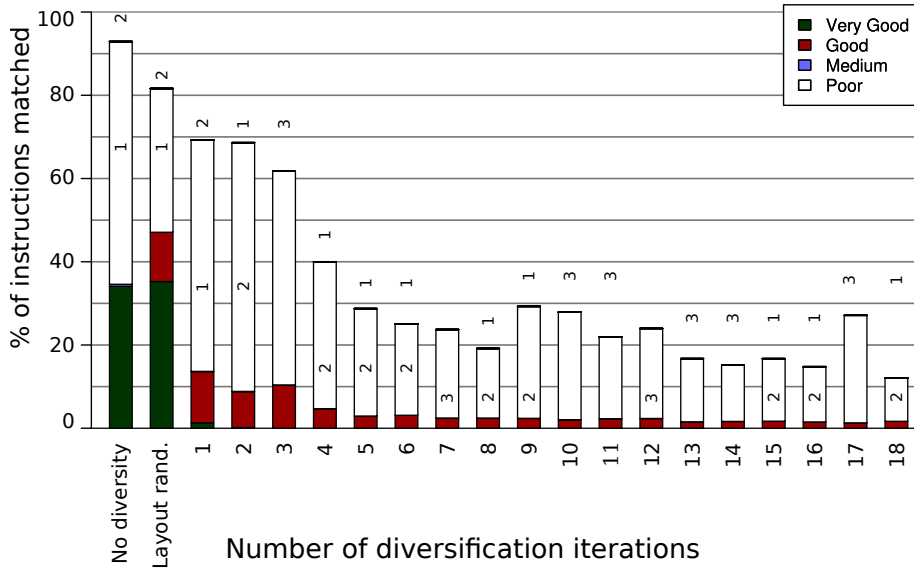


(a) All matches returned by BinDiff

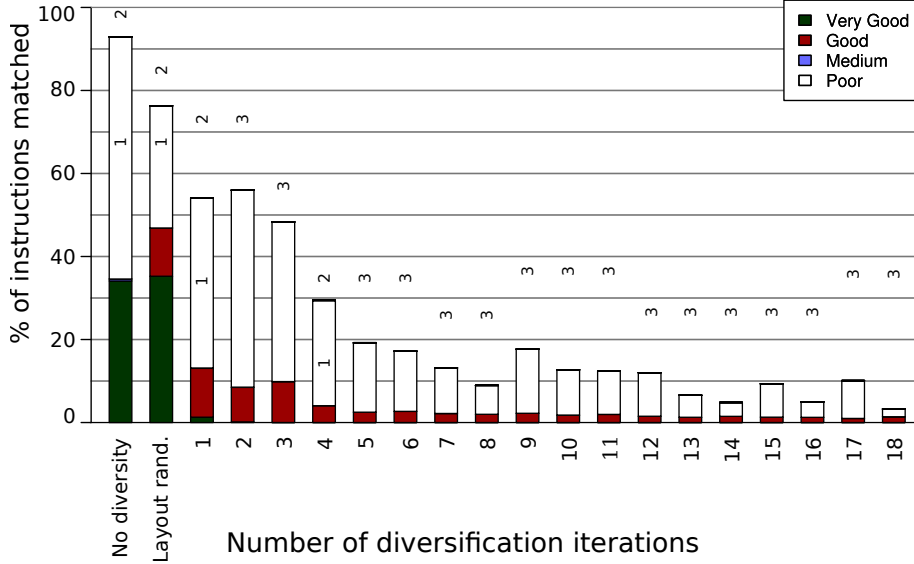


(b) Correct matches returned by BinDiff

Figure 3.3: Diffing results for the `bzip2` use case, indicating the decisive matching heuristics.



(a) All matches returned by BinDiff



(b) Correct matches returned by BinDiff

Figure 3.4: Diffing results for the `bzip2` benchmark, where we grouped the match heuristics by match quality.

the code, and all relevant instructions are in the 85% unmatched code. Clearly, BinDiff is of almost no use to an attacker at this point.

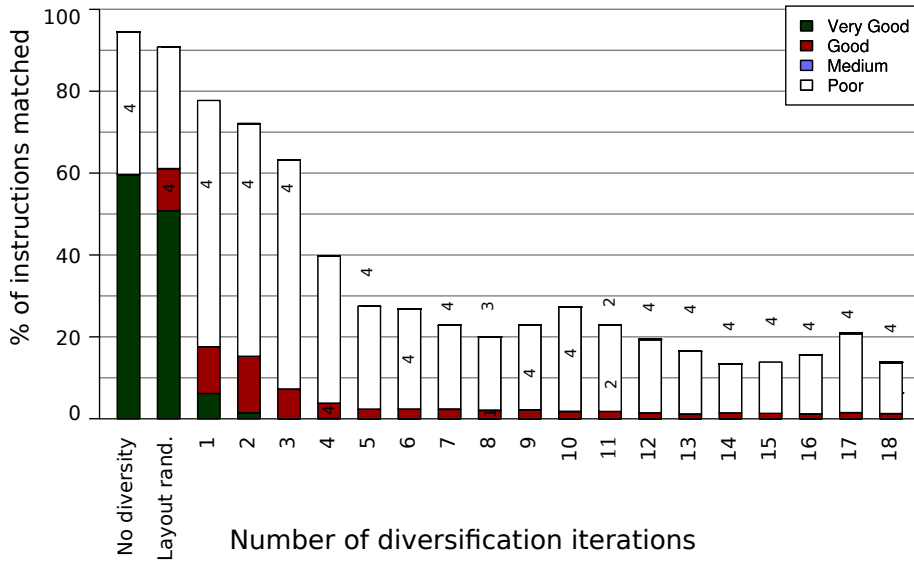
After 18 iterations, an even better result was obtained, but as we shall see, that result was achieved at the expense of more performance overhead.

Figure 3.5 shows the `png_debian` benchmark. Recall from Chapter 2 that the SCIMs in this benchmark are not matched by BinDiff. This can also be seen in this figure: we see that the four changed instructions reside in the functions that are matched with a “weak” matching strategy. After four iterations of Glaucus, the SCIMs are no longer matched. However, at the same time, only 30% of the instructions are matched, as opposed to the 95% without any diversity. Figure 3.6 shows the result for the `png_beta` benchmark, where we observe that part of the SCIMs are matched, while 26 SCIMs are not matched without diversity. After 6 iterations, the SCIMs are no longer considered matched, but now only 20% of the instructions are matched. In Figure 3.7 we see that for the `soplex` benchmark, some SCIMs are incorrectly considered to be matched by BinDiff, even at iteration 17. In this case, we see that the lowest matching rate occurs at iteration 15.

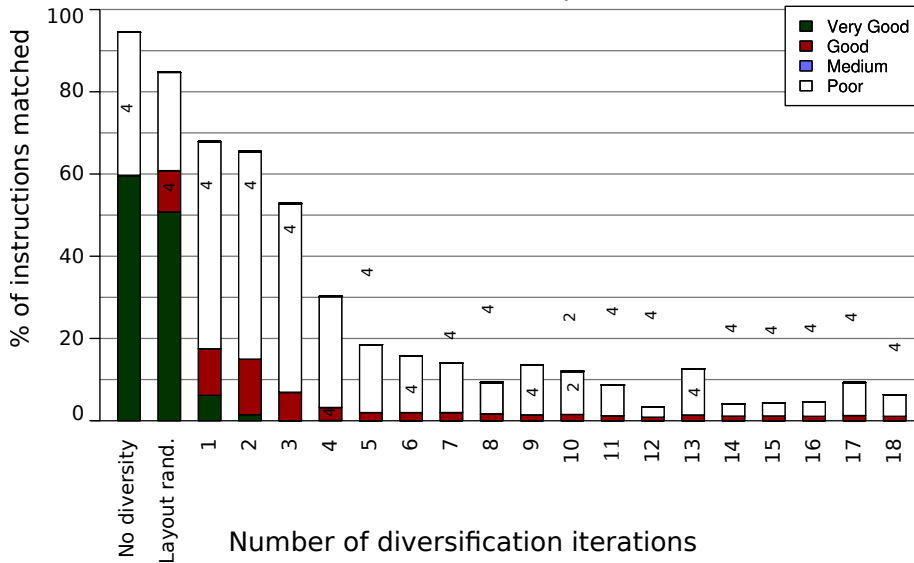
3.3.2 Overhead

Figure 3.8 depicts the overhead of the diversification in terms of code size. Starting with layout randomization, every transformation adds additional code. In iterations 7 and 13, as new transformations become applicable on executed fragments according to the rules in Table 3.1, more overhead is added, compared to the previous iterations. For the most interesting versions of the binaries, the code size overhead is 15 to 25%. As before, whether or not this overhead can be considered acceptable, is a decision that has to be made by the software vendor.

Software distributors typically do not send the entire patched binary. Rather, they use binary patch generation tools to create patches that are smaller to distribute than the entire binaries. For example, Apple, FreeBSD and Mozilla use `bsdiff` to create binary patches for distribution [140]. Figure 3.9 depicts the overhead of diversification in terms of binary patch size when we use `bsdiff` to create the binary patches. The patches become significantly larger, up to the point where their size becomes between 30% and 40% of the full code section size. For isolated patches, such as in the `pngtest_debian` use case, diversifi-

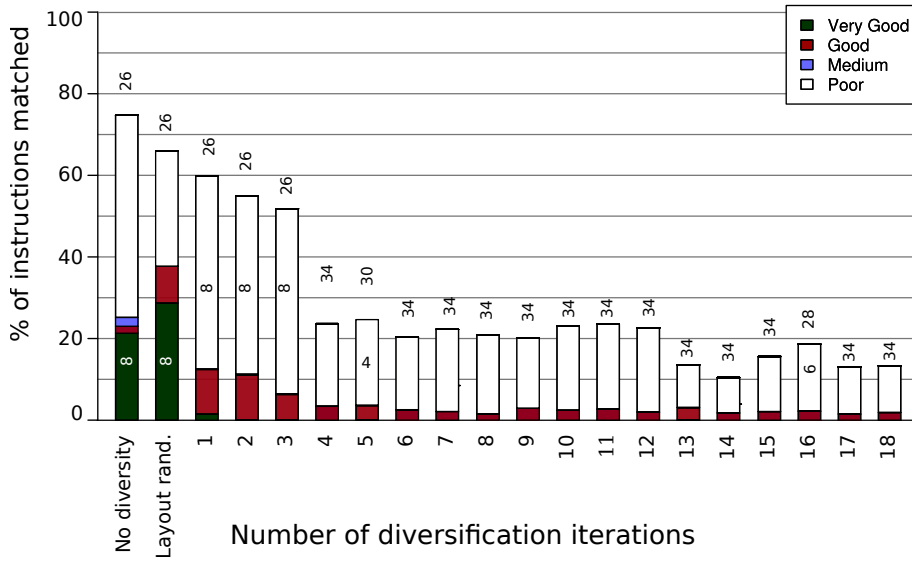


(a) All matches returned by BinDiff

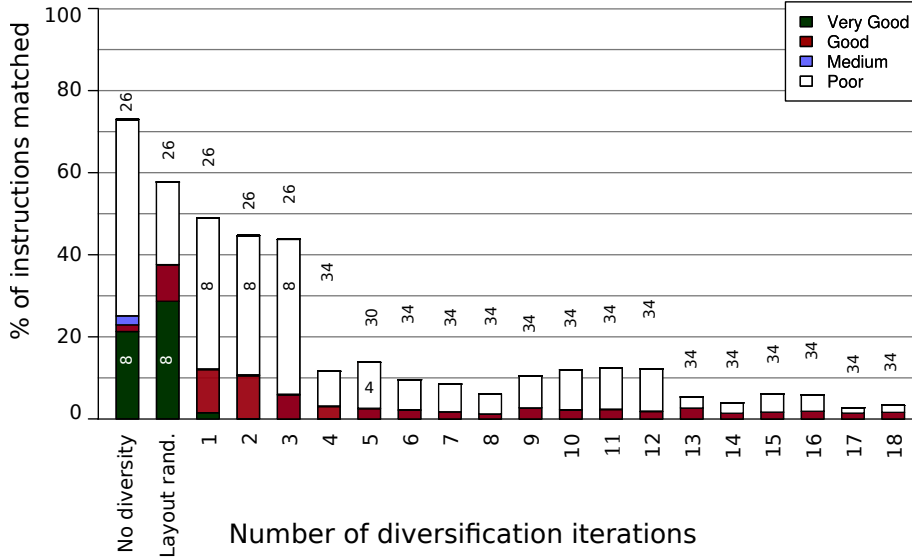


(b) Correct matches returned by BinDiff

Figure 3.5: Diffing results for the `pngtest_debian` benchmark, where we grouped the match heuristics by match quality.

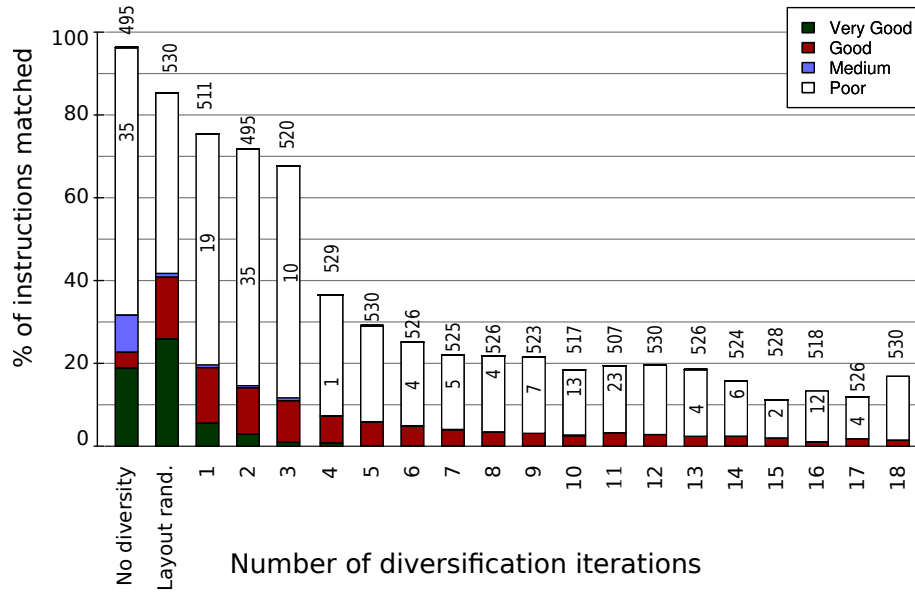


(a) All matches returned by BinDiff

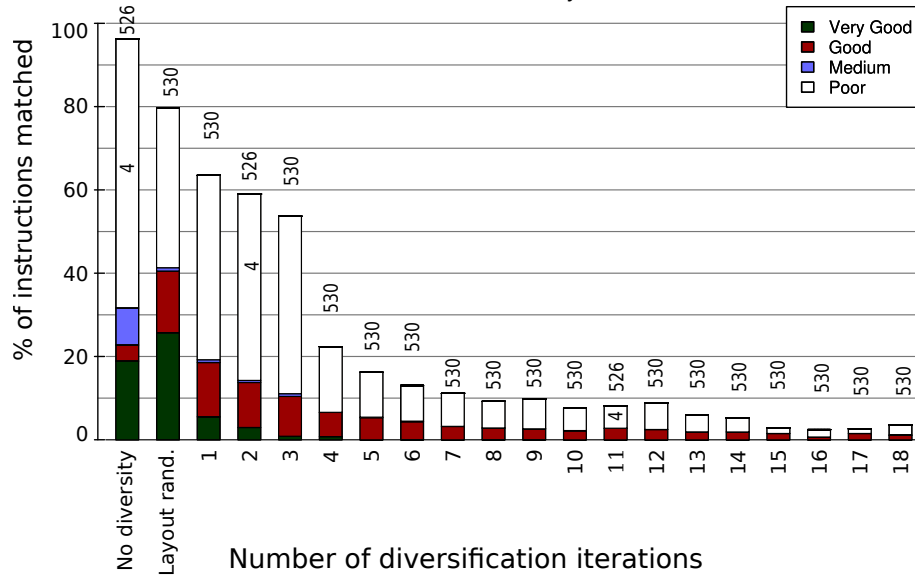


(b) Correct matches returned by BinDiff

Figure 3.6: Diffing results for the `pngtest.beta` benchmark, where we grouped the match heuristics by match quality.



(a) All matches returned by BinDiff



(b) Correct matches returned by BinDiff

Figure 3.7: Diffing results for the `soplex` benchmark, where we grouped the match heuristics by match quality.

cation can increase the binary patch size with a factor 1000. For larger patches, as for `pngtest_beta`, the overhead is limited to a factor 2.5. The overhead to distribute diversified patches, e.g., over the Internet, is therefore extremely variable. The trade-off between this overhead and the provided protection against patch-based attacks is one of the trade-offs that developers will have to make.

For the benchmarks for which we have training and reference inputs from the SPEC benchmark suite, we used the training input set to collect profile information on the patched binary. We evaluated the performance overhead on the transformed binaries with the reference input set. Figure 3.10 presents the performance overhead of our approach.

For `bzip2`, we observe very low overheads until the last but two iterations. The overhead is even negligible for the first 9 iterations. For `soplex`, we initially obtain a low but non-zero overhead, which surpasses 5% as of iteration 13. At that iteration, BinDiff was only able to correctly match about 6% of all code.

When we compare these results with those from Proteus in Chapter 2, we see that we have less execution time overhead, as well as less code size overhead. Furthermore, our increase with execution time overhead actually correlates with a decrease in the pruning rate.

As the plot of slowdown versus diffing success in Figure 3.11 shows, the developer can clearly make a trade-off between protection and overhead. Our approach is able to thwart BinDiff more effectively and efficiently than Proteus. At the same level of execution time overhead and code size overhead, Glaucus reduces the pruning rates for attackers using BinDiff.

3.3.3 Representativeness

Since Glaucus only uses feedback from BinDiff, it is interesting to know how well Glaucus can protect against attackers that use the other IDA Pro-based diffing tools. In Figure 3.12, we see the results of $\mu_{\text{patchdiff2}\backslash\text{extend}}$, $\mu_{\text{TurboDiff}\backslash\text{extend}}$, and $\mu_{\text{BinaryDiffer}\backslash\text{extend}}$ applied to the `bzip2` benchmark. The results for `soplex` are shown in Figure 3.13. We see that both the pruning rate for both `patchdiff2` and `TurboDiff` drops to less than 10% after about four iterations. However, `BinaryDiffer` performs significantly better than those two tools, even after 18 iterations. Even though the pruning rate is less than 50%, it performs

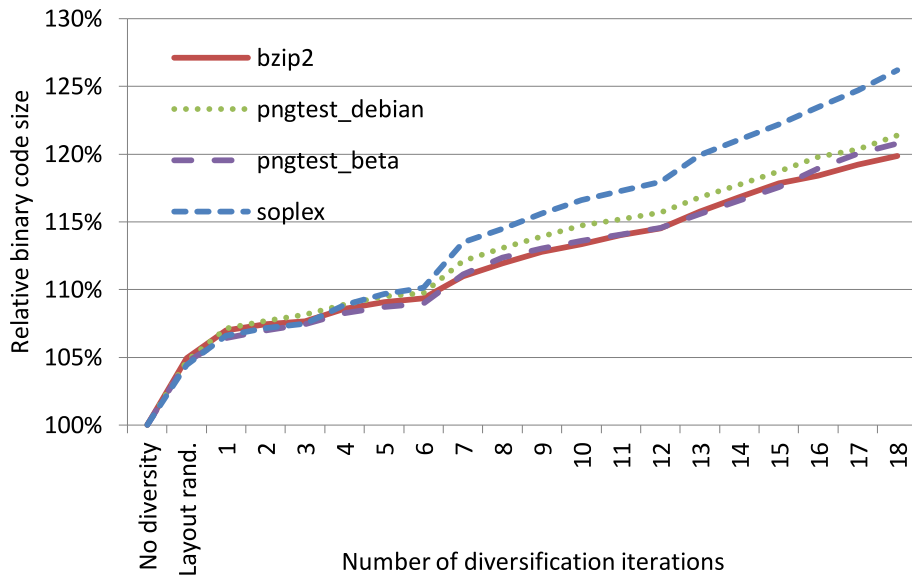


Figure 3.8: Relative binary code size.

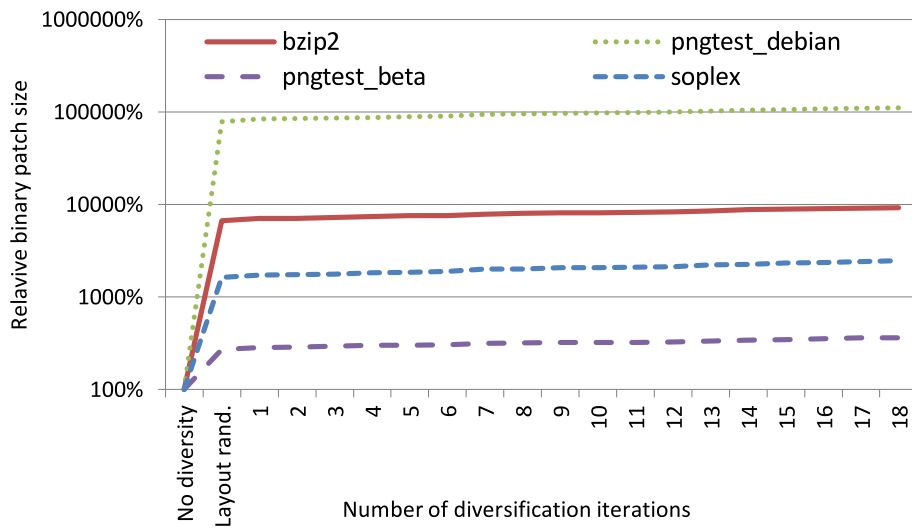


Figure 3.9: Relative binary patch size.

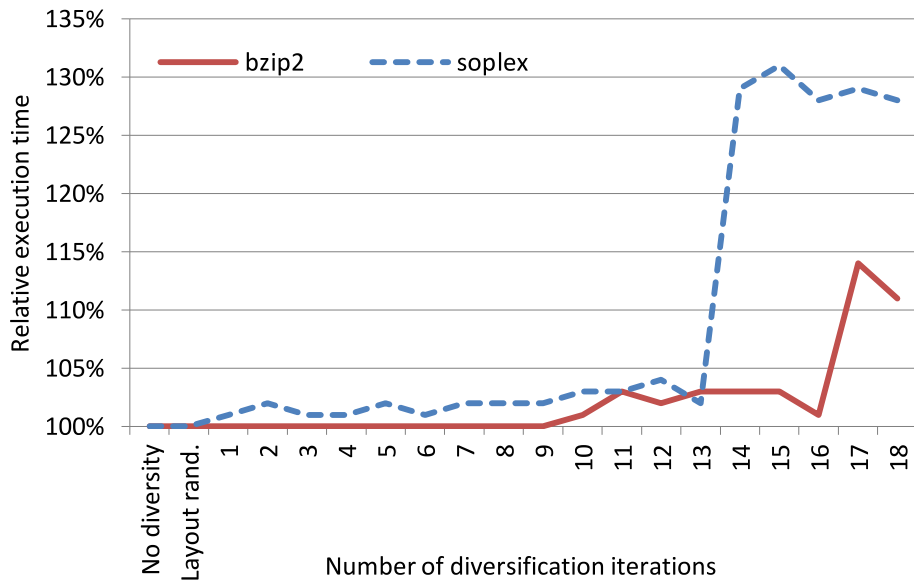


Figure 3.10: Execution times relative to the times of the undiversified binaries.

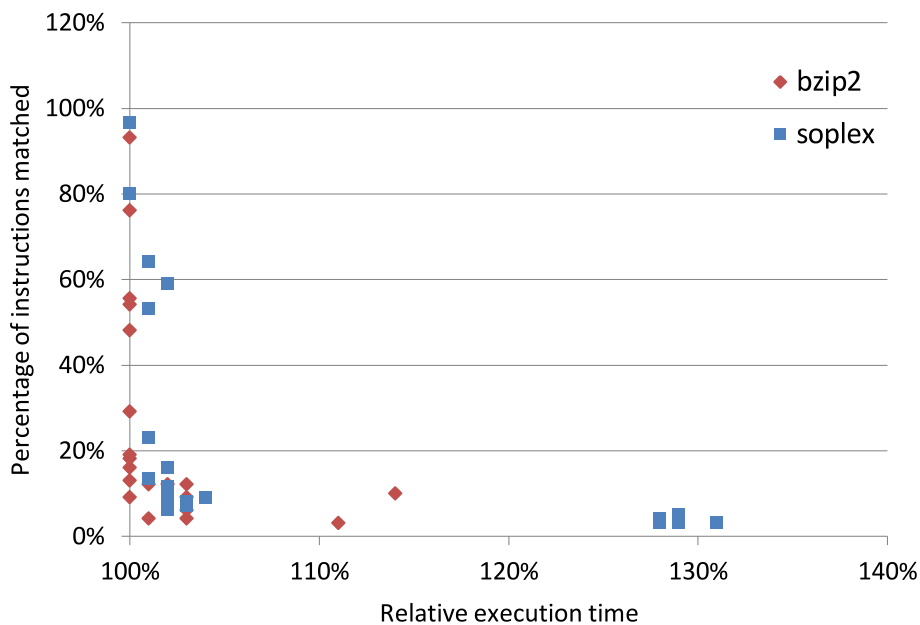


Figure 3.11: Relative execution times vs. percentage of correctly matched instructions.

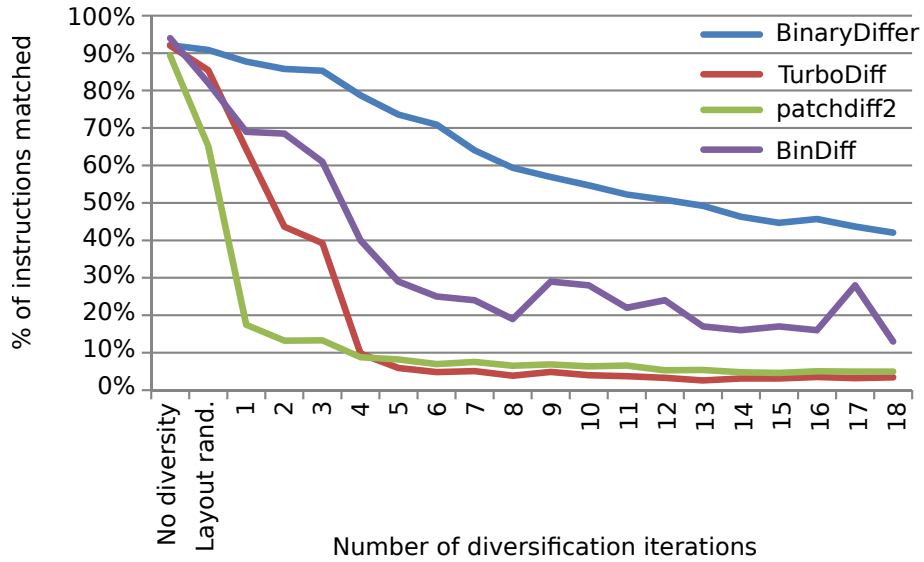


Figure 3.12: Cross-validation of Glaucus with other diffing tools on bzip2.

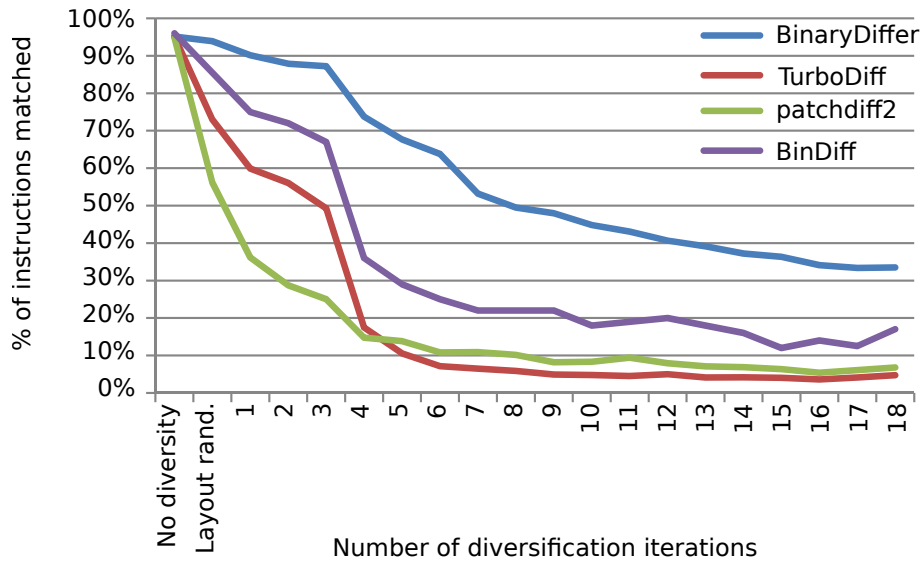


Figure 3.13: Cross-validation of Glaucus with other diffing tools on soplex.

better than BinDiff after the same number of iterations. This means that BinaryDiffer internally uses some matching heuristic that is not as effectively thwarted by our set of transformations. One reason is that BinaryDiffer internally matches BBLs even if they do not belong to a function. The transformations currently implemented in Glaucus mostly focus on changing the structure of functions and thwarting IDA Pro's function construction. Additional transformations can be added to Glaucus that focus on thwarting BBL matching.

3.4 Discussion

The results from Section 3.4 indicate that strong protection against patch-based attacks with BinDiff and other tools can be obtained at a negligible performance overhead, but at a significant cost in patch size. Several issues can still be raised about the proposed approach, however.

First, one may question whether the proposed approach is specific for BinDiff or whether it is more general. Our experiments with other IDA Pro diffing plug-ins, including the results presented in Chapter 2, demonstrate that the approach is effective against all other publicly available plug-ins, but that the effectiveness varies among the different tools. This could be mitigated by including feedback from additional tools. Furthermore, as discussed, it is always possible to extend the attack tools with new analyses and transformations, which could improve the success rate of an attacker. Attackers could try to detect and undo specific transformations, including the obfuscations applied in our approach.

Some of the most effective deobfuscation attacks like dynamic or hybrid static-dynamic deobfuscation [101, 135] are not applicable on some patch types. In such dynamic attacks, the program's execution is first monitored and traced. The traces are then used to remove some of the never-taken execution paths from the program. This builds on the assumption that never-taken execution paths with certain signatures probably originate from obfuscating transformations such as opaque predicates, and hence were not present in the original program. In the case of patches that insert input validation checks that the attacker cannot trigger yet, the traces he collects on the patched program will indicate that the inserted checks are potential opaque predicates. So instead of helping the attacker to obtain a better diffing result, his deobfusca-

tion will cause the SCIMs to be matched incorrectly. Even when such deobfuscation will be successful on *some* kinds of patches, the attacker does not know which kind of patch is being applied, and will thus not know if deobfuscation will make his results useful.

The effectiveness of our approach remains to be studied against more advanced tools such as deobfuscators and code normalization tools. However, this situation is not uncommon in the software protection arms race: once a software protection scheme is in use, attackers will try to break the transformations used. The fact that particular transformations are defeated does not imply that the whole approach using those diversifying transformations is flawed or broken. When attacker tools become more effective, we can extend the set of relatively simple transformations in our current implementation with more complex ones to make attackers require even more complex tools. While the complexity of our current set of link-time transformations is limited because of the lack of high-level semantic information in object files, similar as well as much more complex diversifying transformations can easily be integrated into a compiler. In general, the more complex the tools in the attacker tool box need to become to overcome the protection provided by diversification, the more time-consuming they will be, and hence the smaller the attacker's window of opportunity will become. So we are quite confident that our approach, although it will have to be tuned and extended in the future, provides a solid foundation for protecting against patch-based attacks.

Secondly, it is worth mentioning that our approach can easily be extended to protect successive patch releases. When a patch to v3 is released, protection might be required against collusion attacks against both v1 and v2. In such cases, it is important that an attacker can not learn more by comparing v1 to v3, in addition to comparing v2 to v3. When this is required, it suffices to use a new set of PRNG seeds and to run the diffing tool twice in each iteration to diff v3 against v1 and against v2, and to consider the union of the sets of matched functions in the next iteration.

Thirdly, it is important to discuss some more qualitative, less quantitative types of costs of our approach. In particular, we have to look at the impact our approach has on customer support and code maintenance costs. This depends on the ease with which one can debug the code and interpret bug and crash reports. In this regard, we should point out that our approach so far only involves control flow transfor-

mations. The original code is not rescheduled, register allocation is not changed, and all data layout remains untouched. The latter includes the statically allocated data, as well as the stack (frames) and the heap. As the developers know the mapping between code fragments in the non-diversified binary and in the diversified binary, and all data addresses remain unchanged, we conjecture that debugging the diversified binaries or interpreting crash reports of them is not fundamentally harder. The only requirement is that a map between original code addresses and diversified code addresses is generated and stored. These maps can then be used to interpret the backtraces in crash reports.

3.5 Conclusion

We have shown that we can use the attack models to improve diversification. We introduced the diversification framework Glaucus, that uses feedback from an attack model to iteratively diversify binaries. The output of BinDiff is used to selectively guide the diversification process to code fragments that are matched, while applying no more transformations than strictly needed to code fragments that BinDiff does not match. The transformations applied to individual code fragments are selected to thwart the specific selectors BinDiff used to match these code fragments.

We have shown that with the iterative feedback-driven approach of Glaucus, the performance overhead of the diversified program is less than when using Proteus. At the same time, the attack models show that the attack effort for a patch-based attack increases, compared to attacking programs diversified by Proteus.

Chapter 4

Removing variation in execution time

As discussed in Chapter 1, execution time variation can be an attack vector for software. This execution time variation can be measured, and used to extract secret information from a program. We will discuss how to defend programs against all timing variation caused by differences in the dynamic control flow of a program, and against some timing variation caused by differences in the data flow. We introduce a compiler-based technique that allows developers to remove these timing variations from their compiled program.

We first discuss how we remove control flow (Section 4.1). Next, we discuss how we mitigate some data-flow dependent timing variation (Section 4.2). This results in a timing side-channel aware compiler, whose implementation we describe in Section 4.3. Our compiler is particularly targeted at Intel’s x86 instruction set architecture (ISA). We evaluate our timing side-channel aware compiler in Section 4.4, and discuss its relative merits compared to other techniques, such as removing control-dependent timing using source-to-source transformations in Section 4.5.

4.1 Automatically removing control-dependent variation

As already mentioned in Chapter 1, the execution time of a program depends on its control flow. When the control flow depends on secret data, the attacker can use the variation of the measured execution

time to recover the secret data. To defend programs against attacks that depend on control-dependent timing variation, we will remove the variations in control flow that depend on secret information. We do this by eliminating all control flow transfers that depend on the secret information. The execution will then always be independent of the secret input. All control-flow dependent timing behavior will then be independent of the secret information, and can no longer leak secret information. Indeed, the instruction mix is then independent of the secret information, as are branch prediction, instruction order, data flow dependencies through registers, and instruction fetching from the instruction cache.

The elimination of key-dependent control flow transfers will be performed in a compiler backend. Using such a backend, rather than a source-to-source transformation tool, enables applying the technique more easily to programs written in different source languages. Furthermore, it reduces the risk of the compiler middle-end undoing source-to-source transformations.

We remove control dependencies from the program by rewriting code fragments and break down the problem of rewriting the program into three disjunct sub-problems:

1. *Remove control dependencies from acyclic code fragments.* We explain in Section 4.1.1 how we remove the control flow from code that contains no loops.
2. *Remove control dependencies from cyclic code fragments in function CFGs.* In Section 4.1.2, we describe how to handle code fragments containing loops. To transform loop bodies we will make use of the technique for transforming acyclic code fragments.
3. *Conditional execution of function calls.* Once we have removed the control dependencies from function bodies, calls originating from those functions need to be timing-independent as well. This is described in Section 4.1.3.

4.1.1 Conditional execution of acyclic sequences

For simple, acyclic code fragments not containing any function calls, conditional execution provides an excellent mechanism to get rid of key-dependent control flow. With conditional execution, control flow

dependencies on diverging non-cyclic execution paths such as if-then-else constructs can be transformed into data flow dependencies on a single path.

On architectures that support full conditional execution, either by means of condition flags that activate or deactivate instructions, such as the ARM instruction set [12], or by means of real predicate registers that guard instructions, such as in the case of the TriMedia [144], this transformation is trivial.

We can show how conditional execution works on source code. To conditionally execute instructions, we transform **if** statements. The goal is to make all instructions from the body of the statement execute, independent from the private information. First, we compute the predicate. Then we use this predicate to execute all statements conditionally in the body of the **if**. Consider again the loop body of the modular multiplication algorithm as discussed in Section 1.6.1:

```

4 result = (result*result) % n;
5 if (bit_set(exponent, i))
6     result = (result*a) % n;
7 i--;

```

To transform the statements of this C code into conditionally executed statements, we first compute the condition of the **if** statement, and use it as a predicate for conditionally executing individual statements:

```

4 result = (result*result) % n;
5 c = bit_set(exponent, i);
6 tmp_1 = c ? result * a : tmp_1;
7 tmp_2 = c ? tmp_1 % n : tmp_2;
8 result = c? tmp_2 : result;
9 i--;

```

The lines with the ternary `?:` operator can then be translated into predicated instructions. For example, when we compile this to ARM assembly, making use of conditionally executed instructions (using the division instruction to compute the remainder), compiler could generate code such as the following:

```

1 bl      bit_set      // c = bit_set(...)
2 cmp     r0, #0        // set condition flag to c == FALSE
3 mulne   r0, r6, r8    // if (c) tmp_mul = result*a;
4 udivne  r6, r0, r7    // if (c) tmp_div = tmp / n;
5 mulne   r4, r7, r4    // if (c) tmp_mod = tmp_div * n;
6 subne   r6, r7, r4    // if (c) result = n - tmp_mod;
7 sub     r1, r1, #1    // i--;

```

The conversion from code with branches to code with conditional execution is called *if-conversion* [6], and it is well known in the field of compiler techniques, in particular for VLIW and EPIC types of architectures. Composing more complex conditional structures such as nested if-then-else structures poses no problem. It only requires additional instructions to compute the nested predicates or to set the correct condition flags, depending on the expressiveness of the architecture with respect to conditions [15].

However, this technique is not directly applicable to ISAs that support only limited or no predicated instructions. On Intel's x86 ISA, only some `mov` instructions can be executed conditionally. This can be mitigated for some instructions by converting them from conditionally executed instructions to unconditional ones that act on temporary variables. The computed value is then conditionally moved to the actual target variable. If we then transform the above source fragment, we get the following code fragment:

```

1 result = (result*result) % n;
2 c = bit_set(exponent, i);
3 tmp_1 = result * a;
4 tmp_2 = tmp_1 % n;
5 result = c? tmp_2 : result;
6 i--;

```

This predicate elimination, also called predicate speculation [133] is fine when the instruction whose predicate is eliminated only affects local variables and state that does not define the global program state, or when the instruction causes no other side effects such as exceptions. For example, in the code fragment above, the unconditional execution of the multiplication in `tmp_1` poses no problems. However, unconditionally executing the division instruction can lead to problems. Consider the following example:

```

1 if (count != 0)
2     average = sum / count;
3 else
4     average = 0;

```

As we did before, we could try rewriting it as follows:

```

1 c = count != 0;
2 tmp_1 = sum / count;
3 average = c ? tmp_1 : average;
4 tmp_2 = 0;
5 average = !c ? tmp_2 : average;

```

In the original code, no division by zero can occur. In the transformed code, however, we *do* divide `sum` by `count` even when `count == 0`, triggering a division-by-zero exception. Thus, the transformed code does not exhibit the same behavior as the original program, and is incorrect. As mentioned earlier, the x86 ISA only supports conditional move instructions, so we cannot fix the faulty transformation by executing the division conditionally. So when we remove all control dependencies on the x86 architecture, we have to rely on some other transform for instructions with side effects.

Our general solution to remove control dependencies on ISAs with limited predication support, to be applied by the compiler backend, consists of the following:

1. Before merging a path into other paths with if-conversion, ensure that all instructions without side-effects in this path operate on local, temporary variables. This can be easily done when using single static assignment (SSA) for representing the instructions [110]. SSA is often used as an intermediate representation in compilers. In SSA code, all instructions write into a new (virtual) register. This means that all instructions already operate on a register when using most modern compilers.
2. Separately transform the instructions that may cause exceptions or that change the global state (for example because they store something to memory). We do this by adding so-called *operand selection safe-guard instructions*. These instructions conditionally select a safe value that is used as input for the transformed instruction. The safe values are chosen so that they do not cause the instruction to cause exceptions or to change the program state. When the predicate for executing the transformed instruction is true, the original input to the transformed instruction is used.
3. Merge the resulting code paths as in regular if-conversion.

Consider the following example code fragment:

```
1  if (c) {  
2    *a = 10;  
3    d = x/y;  
4  } else {  
5    b = 10;  
6  }
```

Using the three steps discussed above, we rewrite this code fragment into:

```
1 tmp_a = a;
2 if (!c) tmp_a = dummy_location;
3 *tmp_a = 10;
4 tmp_y = y;
5 if (!c) tmp_y = 1;
6 tmp_d = x / tmp_y;
7 tmp_b = 10;
8 if (c) d = tmp_d;
9 if (!c) b = tmp_b;
```

If `c` evaluates to false, the store operation will now write to a dummy location where it does not harm the real program state, and the division will be executed with divisor 1, which will not cause an exception. Furthermore, in that case only the assignment to `b` will be executed, but not the one to `d`. If `c` evaluates to true, the store and the division will be executed as they should, and only the assignment to `d` will be executed, not the one to `b`.

The most common instructions that should be safe-guarded this way are divisions, loads, and stores. For loads and stores, we allocate a memory location on the stack that is used as dummy value. An interesting question is whether or not the execution of loads or stores often depends on secret keys. The fact is that big integer functionality for cryptographic purposes, such as RSA, is usually implemented using memory arrays of smaller integers. When computing the result of an operation on such big integers, the final results, as well as the intermediate results, have to be written to memory into these arrays. So in such libraries, conditional loads and stores are as likely as any other conditional operation.

Note that if-conversion should only be applied when the involved conditional branches depend on a secret key. Whether or not the branches depend on secret information can be determined by means of a compiler data flow analysis. However, such analyses are typically not sufficient. The output of an encryption algorithm depends on both the secret plaintext and the secret key, but the output of the encryption itself is generally not considered to be secret information. Information about when values depending on secret information is itself no longer secret, can be taken into account with custom *declassification properties* [127]. In case the compiler lacks the required precision to compute the correct dependencies, the compiler should either conservatively as-

sume that there is a dependency on the secret key, or user annotations of the source code could clarify this for the compiler. In any case, the user should already inform the compiler what exactly constitutes the key from which control flow should be made independent, for example by means of so-called *attributes* or by means of *pragmas* [39, 68].

Finally, we should point out that our implementation using conditional execution of `mov` instructions relies on the fact that conditional moves do not cause data-independent timing behavior. We discuss this assumption in detail in Section 4.4.

4.1.2 Cyclic control flow graphs

Cyclic control flow graphs occur for program fragments containing loops. In these loops, the number of iterations may be key-dependent or not.¹ A compiler can again either detect this by means of static data flow analysis, or it can rely on user-annotations in the program.

For cryptographic code that resists side-channel attacks, loops should have a number of iterations that does not depend on the value of a secret key. However, cryptographers and programmers do not always keep this in mind, which makes the code possibly leak secret information. Furthermore, some cryptographic libraries are programmed generically, e.g., for different key lengths, so that the number of iterations of certain loops depends on the length of the key. In the latter case, user-annotations specifying that a loop's number of iterations is not dependent on a key will be required. The compiler could aid the user by giving warnings about loops for which it requires, but cannot find, a user annotation.

When a loop's iteration count does depend on the actual value of a secret key, the compiler could try to determine (or the programmer should specify) an upper bound on the number of iterations. The iteration count of the loop can then be fixed to this fixed upper bound, and the loop body will be executed conditionally: a separate condition will keep track of whether real iterations are still being executed, or whether additional iterations are being executed to reach the fixed upper bound.

Please note that from a performance point of view, increasing the number of iterations in a loop to a fixed upper bound might be detri-

¹In lower-level code, this corresponds to a conditional branch that determines whether the loop is continued or exited. Such a branch, if taken, transfers control back to the loop entry point. As such, it cannot be omitted with simple if-conversion.

mental. However, this is unavoidable. The best we can hope for is getting a fixed execution time that is equal to the slowest time of the original program. If this slows down the program on what originally was its best-case performance input, so be it.

To rewrite a loop, we consider only loops with a single back edge, which furthermore only have a single loop exit edge. The back edge and the exit edge originate in the same basic block. Loops that do not satisfy these constraints, are first restructured. When we remove the control dependencies from a loop, we need to transform the loop body and the back edge's control transfer.

When the control transfer does not depend on the private information, we can just leave the control transfer unaffected.

In the other case the control transfer depends on the private information. As mentioned above, we assume that we have a static upper bound on the number of iterations of the loop. Thus, the compiler can transform the loop control into a simple counter with a bounds check that is independent of the private information. The loop body is then conditioned on the original control. For example, when we have a key-dependent loop such as the following:

```
1 for (int i = 0; i < key->nr_bits; i++) {
2     key->bits[i] = 0;
3 }
```

We can transform this code as follows, assuming a static loop upper bound of 16:

```
1 int execute_body = TRUE;
2 for (int i = 0; i < 16; i++) {
3     execute_body = execute_body && i < key->nr_bits;
4     if (execute_body) key->bits[i] = 0;
5 }
```

Up until now, we discussed loops that are entered unconditionally. However, loops can be entered conditionally as well. We can easily extend the above transformation to deal with conditionally entered loops. When we remove control dependencies from acyclic control flow with if-conversion, we keep track of a predicate that signifies whether or not instructions need to be executed. We also use this predicate to conditionally execute a loop body. Rather than unconditionally setting `execute_body` in the code fragment above to `TRUE`, we can initialize it with the predicate on which the entire loop's execution is predicated. Thus, we can transform


```
1  if (key->valid) {
2      for (int i = 0; i < key->nr_bits; i++) {
3          key->bits[i] = 0;
4      }
5  }

into

1  int execute_body = key->valid;
2  for (int i = 0; i < 16; i++) {
3      execute_body = execute_body && i < key->nr_bits;
4      if (execute_body) key->bits[i] = 0;
5  }
```

Note that we introduced new control dependencies when transforming the loop body. Furthermore, even the untransformed loop body could already contain control dependencies. These need to be removed to remove the variation in execution time due to control flow. The loop body can also contain nested loops with their own control dependencies. To remove all these control dependencies, we do a structural recursion on the loop as follows:

- *The loop body is an acyclic code fragment.* This is the base case for the recursion. We just apply if-conversion to the loop body as described in Section 4.1.1.
- *The loop body itself is cyclic.* We recursively apply the technique from this section. The predicate for the execution of the nested loop body will then not only depend on the nested loop condition, but also on the outer loop's execution predicate.

4.1.3 Function calls

Any realistic program contains functions and hence function calls. In particular, big integer libraries usually represent operations on big integers with function calls, so function calls will also occur in cryptographic code. These function calls will also need to be executed conditionally.

A first solution to this problem constitutes inlining. When a callee's function body is inlined in a caller's body, the call is removed and the callee's body can be executed conditionally like the caller's code.

Because inlining may increase the code size of a program significantly, which may not be acceptable for embedded devices with small

amounts of memory, an alternative solution is required. Our solution adds an additional function parameter that specifies whether or not the function would have been invoked in the original program. Instead of executing a call conditionally, each call will now be executed unconditionally with the additional parameter. Inside the function, all code is then executed conditionally based on the additional condition specified by the added parameter.

For example, consider the following code fragment:

```
1 void f(int x) {
2     *a = x;
3 }
4
5 int main(int argc, char** argv) {
6     ...
7     if (c)
8         f(10);
9     ...
10 }
```

This will be converted into

```
1 void f(int c_f, int x) {
2     if (c_f) *a = x;
3 }
4
5 int main(int argc, char** argv) {
6     ...
7     f(c, 10);
8     ...
9 }
```

Here we used the previously described techniques for removing control dependencies on the function `f()`.

Please note that in practice, we duplicate a function like `f` and we add the additional parameter to the duplicate only. That way, any call to `f` that is not dependent on the secret key can still invoke the original function version without the additional parameter, which will be better for performance and which requires less code to be adapted.

Function calls can be either direct or indirect. Examples of indirect function calls are function pointers and virtual method calls like those in C++ or Java. While we only implemented our technique to deal with direct function calls, our technique can also be extended to indirect function calls. We could store information on whether or not this

indirectly called function needs to be executed conditionally in global, thread-local storage, instead of passing it as an argument. The transformed function can use this information from the thread-local storage for conditionally executing its instructions. This way, we do not change the function's signature, and a program can mix indirect calls to functions that have been transformed and functions that have not been transformed.

Function pointers are data values that can be key-dependent as well. Calls to such key-dependent functions need to be constant-time as well. A pointer-analysis could determine to which functions this pointer can refer. All possible functions could then be transformed so that they all have the same execution time by combining all function bodies into a single function. However, this could introduce a significant performance overhead. Furthermore, like conditionally executed memory operations can point to unallocated memory, so can conditionally called function pointers point to invalid code addresses. For such cases, we could provide a dummy function pointer to be called similarly to how conditional stores and loads are handled.

4.2 Removing data-dependent variation

As discussed in Chapter 1, instructions exist whose execution time depend on their operand values. Such instructions need to be considered as well when designing a timing side-channel aware compiler. We considered two kinds of instructions with a variable execution time. First, we discuss instructions whose execution time depends on their algorithmic implementation in the processor. We use the division instruction as our test case for this class. The second kind of instruction are instructions that use the memory subsystem.

4.2.1 Timing variation due to early exit

Similar to how programmers can introduce variation in execution time by introducing control flow dependencies, so can hardware designers for the execution time of a *single instruction*. For example, the execution time of an instruction to multiply or to divide two numbers can depend on the number of leading zero-bits in the operands. Such a behavior is also called *early exit*. As an example, we focus on the `div` instructions of Intel's Core 2 processors, whose execution time depends

on both operands on the processor versions we evaluated [41]. Similar techniques could be used against ARM processors whose multiplication instruction has a variable execution time.

Processor designers can add options that configure the dependence of the execution time of instructions on their operands. For example, ARM processors optionally allow software to make multiplication instructions operate in constant time [13]. Intel could opt to add a similar configuration option for the execution time of its division instructions. Compilers that generate code for such processors can make use of such a configuration option to make code execute in constant time regardless of operand values for multiplications and divisions. However, since Intel processors offer no such option, our compiler will have to deal with each sensitive division instruction individually. The most general approach our timing side-channel aware compiler supports, is to emulate division instructions by calls to constant-time functions that perform the same operation. We can easily extend this approach to architectures that have similar variable-execution time instructions.

A downside of this approach is its significant overhead. To avoid this, we can try to rewrite the program's use of the variable latency instruction in such a way that the total execution time is constant, regardless of the secret inputs to the code fragment containing the instruction. We evaluated different ways in which occurrences of the division instructions can be rewritten [143]. The full details of this research will be published in the PhD work of Jeroen Van Cleemput. However, we will briefly summarize those results here.

As a first approach we investigated to make the execution time constant, by adding constant-time compensation code, which is executed in parallel with the division instruction. We let every use of the division instruction's result depend on the division *and* the compensation code. This is shown in Figure 4.1. The idea is that the execution time of the compensation code exceeds the worst-case execution time of the division instruction. The execution time of the resulting program should then depend only on the execution time of the compensation code, which is constant. However, we observed that such schemes do not work in practice. The complex, out-of-order behavior of the Intel CPUs on which we tested these schemes thwarts attempts to make the execution time of transformed programs constant. While we could generate some programs in which the total execution time was constant, whether or not a particular code fragment had a constant execution

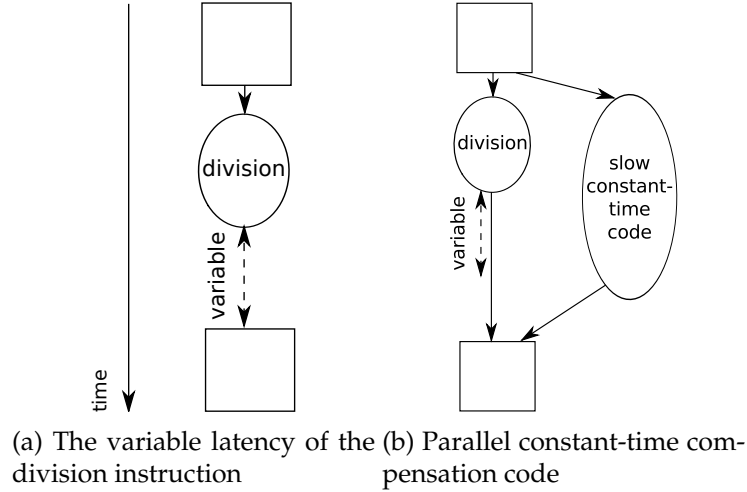


Figure 4.1: Data dependency graph of parallel constant-time code to make the result of divisions constant time as well. When adding parallel constant-time compensation code, the end result depends on both the variable latency division and the slower, constant-time code.

time depended heavily on the surrounding code and the compensation code. We could not predict with which compensation code the total execution time would be constant.

We could however achieve reliable, constant execution times by executing the division instruction multiple times. We observe that the execution time of the division instruction only takes 6 discrete values on the core version we evaluated. We call each of these distinct execution times a time class. For other core versions, the number of time classes can be different. The operands to the division can be classified according to which of these time classes their resulting division belongs to. We insert code that computes this classification at run time. This is followed by a division for *all* possible execution times, where we use a similar approach to our dummy location values: we have dummy operands for each time class, and choose dynamically between those dummy operands and the original operands to the division. This is illustrated in Figure 4.2.

4.2.2 Timing variation due to the memory subsystem

Emulating the functionality of instructions is not always possible. In particular, we cannot emulate instructions that access the program's

```

1 // Which execution time has this division?
2 class = divisor < 2      ? 1 :
3     divisor < 0x20      ? 2 :
4     divisor < 0x200     ? 3 :
5     divisor < 0x2000    ? 4 :
6     divisor < 0x20000   ? 5 : 6;
7 // use a fixed-latency bit-scan-reverse (bsrl) operation
8 leading_zeroes = 31 - bsrl(dividend);
9 dividend <<= leading_zeroes;
10 // Execute 1 division with fixed time per latency class:
11 result1 = dividend / ( class == 1 ? divisor : 0x2);
12 result2 = dividend / ( class == 2 ? divisor : 0x20);
13 result3 = dividend / ( class == 3 ? divisor : 0x200);
14 result4 = dividend / ( class == 4 ? divisor : 0x2000);
15 result5 = dividend / ( class == 5 ? divisor : 0x20000);
16 result6 = dividend / ( class == 6 ? divisor : 0x200000);
17 // Select the correct result:
18 quotient = class == 1 ? result1 :
19     class == 2 ? result2 : ...;
20 quotient >>= leading_zeroes;
21 remainder = dividend - (quotient * divisor);

```

Figure 4.2: C code equivalent of unsigned division computation using only fixed-latency divisions. All selection statements $a ? b : c$ are implemented with fixed-latency conditional moves. The (re)computation of the remainder is necessary because the remainders computed using shifted dividends are not correct. Similar code can be used for signed division, but then the sign needs to be corrected afterward.

memory to not require memory access. However, we can still try to mitigate some of the variation introduced by the memory subsystem.

Variation due to cache behavior As discussed earlier, processors have data caches to speed up accesses to program memory. However, this introduces variation in the execution time of programs.

The best solution would be to modify the behavior of the processor's data caches, so that the caches no longer leak information through their timing behavior. One way would be to disable the caches entirely [82], which would however have a serious impact on performance. Another approach would be to use specialized cache architectures that focus on security. These will not leak information through timing. An example of such cache architectures are partitioned

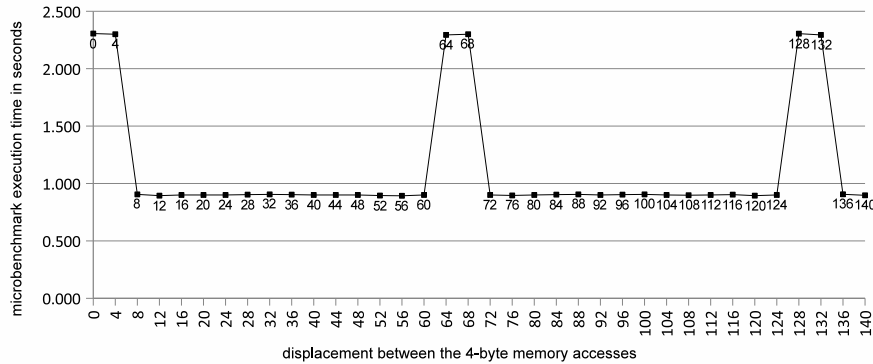


Figure 4.3: Execution times of a microbenchmark loop with 4-byte load and store instructions executed for varying displacements between the accessed locations.

caches [153]. However, such cache architectures are not currently available in x86 processors. In this work, we do not try to resolve this problem on x86 processors.

Load bypassing Even when the instructions always access data in the cache, there remain possible sources of timing variation. We investigated the potential timing variation due to load bypassing.

Consider the following loop body:

```

1  loop:  mov dword [ebx], 2
2         add eax, [ecx]

```

This fragment will write to the memory location specified in `ebx` and read from the memory location specified in `ecx`. If this fragment is executed in a loop with enough iterations, the cache behavior will not influence the timing behavior significantly because only two memory locations are touched.

We measured the timing behavior of such a loop on an Intel Core 2 Duo, where we load and store 4-byte `int` values. We varied the displacement between `[ebx]` and `[ecx]` over a large range in steps of 4 bytes. The resulting execution times are depicted in Figure 4.3. When the store accesses the same data as the load (displacement 0), a higher execution time is seen than when, e.g., there is a displacement of 12 bytes between the two accesses. The reason is a micro-architectural feature called load bypassing [136]. When the store instruction and the load instruction access the same memory location, the out-of-order

processor detects a data dependency between them by comparing the addresses of the locations at which they access the memory. This dependency forces the processor to let the load wait for the store. When the address comparison indicates that there is no such dependency, as when the displacement is 12 bytes, the load can be executed together with the store without having to lose time waiting.

Surprisingly, we observe the same slowdown whenever the displacement modulo 64 is in the range $[0,7]$. When we repeated the experiment for loads and stores of other widths, such as with 1-byte `char` accesses, we observed a similar slowdown. This illustrates that even independent memory operations that always hit in the cache can still cause timing effects as if they were dependent. These effects are not related to the particular combinations of cache lines or cache banks that are accessed. Instead these effects are caused by so-called *pessimistic load bypassing* [136]. Pessimistic in this context means that the processor only compares a few bits in the memory addresses to decide whether or not two memory accesses depend on each other. As soon as those bits are identical, the processor pessimistically concludes that there may be a data dependency. The execution is then slowed down, even when there is in fact no need to.

For our purpose of defending against timing attacks, this observation indeed confirms that even if we can avoid all timing dependencies on cache behavior, we still need to take care of other data dependencies through memory. These dependencies can be true data dependencies through memory, but they can also be dependencies between seemingly independent operations that are caused by micro-architectural pipeline implementation details.

We were able to solve this problem by inserting no-operation instructions between the store and load operations. Once we insert five or more no-ops, the resulting curve of execution times became indistinguishably flat. We applied t-tests [63] to show that the resulting distributions indeed became indistinguishable. For other similar code fragments, similar looking results were obtained, indicating that with enough no-ops inserted where needed, this time side channel can be closed. As with the mitigation technique for division instructions, a full analysis of this mitigation strategy will be published in the PhD work of Jeroen Van Cleemput.

4.3 Implementation

In order to make an implementation of our proposed technique of making code fragments execute in constant time practical, we need to present it to programmers in a usable way. If a programmer has to run a lot of different tools, or has little choice on how the technique is applied, it cannot be considered really useful. As a proof-of-concept, we implemented our technique as an extension for the LLVM compiler framework² [94]. The LLVM framework allows people to write custom analyses and transformation passes. These can either be integrated into the LLVM framework, but they can also be provided as plug-ins so that users can easily combine the required transformation with their own, custom analyses and transformations, without having to use or adapt their code to a custom compiler. We opted for implementing our technique as a plug-in.

In our proof-of-concept implementation, we implemented support for annotations to let the programmer decide which functions need to be transformed. The simple relatively form of annotations we have implemented enables the programmer to specify which C functions must have data-independent control flow. In our implementation, we call this the control flow to be balanced, hence we add the attribute “balanced” to the function. Consider the following code fragment:

```
1  int __attribute__((annotate("balanced")))  
2  f(int a) {  
3      if (a > 0)  
4          return 1;  
5      else  
6          return -1;  
7  }  
8  
9  int g(int* a) {  
10     if (a!=0)  
11         return *a;  
12     else  
13         return 0;  
14 }  
15  
16 int __attribute__((annotate("balanced")))  
17 h(int a, int* b) {  
18     if (a & 2)
```

²<http://www.llvm.org/>

```
19     return f(a);  
20 else  
21     return g(b);  
22 }
```

In this fragment, functions f and h will be adapted, but g will not. However, because h calls g , an adapted version of g with data-independent control flow will be generated which will then be called from within h .

Our current implementation lacks support for recursive function calls and for loops with a variable execution count, but adding support for those is merely an implementation issue that poses no fundamental challenges. Furthermore, that support is not needed to evaluate the effectiveness and efficiency of our technique.

4.4 Evaluation

To evaluate our approach and the extent to which the x86 architecture lends itself to executing timing-variation free code, we protected a number of microbenchmarks against timing-based side-channel attacks.

4.4.1 Experiments

We used a variety of microbenchmarks to evaluate the efficiency and effectiveness of our proposed approach. Efficiency here corresponds to the performance and code size, i.e., the execution time overhead and code size overhead introduced by the if-conversion and elimination of variable-latency division instructions in the protected software. Effectiveness corresponds to the extent with which the protection is able to make the timing behavior independent of sensitive data such as secret keys. The C source code of these microbenchmarks appears in Figure 4.4. We measured this effectiveness by measuring the differences in execution times for different, carefully selected inputs, both before and after our code transformations. We selected the inputs so that the total amount of executed instructions differs significantly, making it easier to measure the variation in execution time. To demonstrate that our technique can defend against side-channel attacks based on branch predictor behavior [1], we measured differences in branch prediction behavior. Both measurements were performed using performance counters.

Code path benchmarks:

```

1  int __attribute__((annotate("balance")))
2  f1(int a, int b, int c, int d) {
3      return a+b;
4  }

1  int __attribute__((annotate("balance")))
2  f2(int a, int b, int c, int d) {
3      if (a < b)
4          return a+b;
5      else
6          return c+d;
7  }

1  int __attribute__((annotate("balance")))
2  f3(int a, int b, int c, int d) {
3      if (a < b) {
4          if (c < d)
5              return c+d;
6          else
7              return c-d;
8      } else {
9          if (a > d)
10             return a-d;
11          else
12             return a+d;
13      }
14  }

1  int __attribute__((annotate("balance")))
2  f4(int a, int b, int c, int d) {
3      if (a < b) {
4          if (c < d) {
5              if (a < 0)
6                  return c+d;
7              else
8                  return c-d;
9          } else {
10             if (b < 0)
11                 return b+c;
12             else
13                 return a+b;
14          }
15      } else {
16          if (a > d) {
17              if (d < 0)
18                  return a-d;
19              else
20                  return a+d;
21          } else {
22              if (c < 0)
23                  return c+a;
24              else
25                  return c-a;
26          }
27      }
28  }

```

Memory benchmarks:

```

1  int __attribute__((annotate("balance")))
2  memread1(int a, char* b) {
3      if (a == 0) {
4          return *b;
5      } else {
6          return a;
7      }
8  }

1  int __attribute__((annotate("balance")))
2  memread2(int a, char* b) {
3      if (a == 0) {
4          return b[0] + b[1];
5      } else {
6          return a;
7      }
8  }

```

OpenSSL fragment:

```

1  for (i = min; i != 0; i--){
2      t1= *(ap++);
3      t2= *(bp++);
4      if (carry) {
5          carry=(t1 <= t2);
6          t1=(t1-t2-1)&BN_MASK2;
7      } else {
8          carry=(t1 < t2);
9          t1=(t1-t2)&BN_MASK2;
10     }
11     *(rp++)=t1&BN_MASK2;
12 }

```

Figure 4.4: Microbenchmarks on which we evaluated our timing side-channel aware compiler.

The first set of microbenchmarks consist of a set of simple C functions of increasing complexity. This set allows us to determine the cost of protecting increasingly complex code. Four microbenchmarks `f1`, `f2`, `f3`, and `f4` contain an increasing number of nested `if-then-else` constructs, and thus an increasing number of different execution paths. Two microbenchmarks `memread1` and `memread2` contain memory accesses that have to be safe-guarded with dummy addresses.

The second set of microbenchmarks consists of three hand-written implementations of modular exponentiation (similar to the code of Chapter 1) as it occurs in, e.g., RSA encryption:

1. The experiment **modexp32** uses 32-bit numbers for a modular exponentiation.
2. The experiment **modexp64** uses 64-bit numbers for a modular exponentiation. This is the native word width on 64-bit platforms such as the Intel Core 2 Duo machine we used for testing.
3. We implemented a minimal so-called big integer component **modexp256** that computes the modular exponentiation of 256-bit integers. This is implemented in C++, using calculations on integer numbers represented as a sequence of 32-bit integers. These 32 bit integers are stored in memory rather than in registers.

These microbenchmarks thus represent the following code:

```
1 result = 1;
2 i = log2(exponent);
3 do {
4     result = (result*result) % n;
5     if (bit_set(exponent, i))
6         result = (result*a) % n;
7     i--;
8 } while (i >= 0);
```

When these microbenchmarks are compiled for 32-bit and 64-bit integers, the compiler maps the modulo computation onto x86 division instructions.

To demonstrate the effectiveness of our approach, we ran these three modular exponentiation microbenchmarks on inputs consisting of (1) random modulo values, (2) random base values, and (3) four different types of exponents.

- In the **all zero** input set, the exponent in binary format consists of all zeroes, except for the two most-significant bits that are set to one. This ensures that the variable `result` (see the modular exponentiation code fragment in Chapter 1) does not remain constant throughout the whole loop. Having all other bits set to zero ensures that the conditional code in the original loop will only be executed twice per loop. This pattern results in very accurate branch prediction by the processor, and the fewest executed instructions.
- In the **all one** input set, all bits in the exponent are set to one. This ensures that the conditional code in the loop is executed in every iteration. So in total, the conditional code is then executed 32/64/256 times per loop for 32/64/256-bit numbers. This pattern also results in very accurate branch prediction by the processor. So when this input is fed to a benchmark, much more code is executed than with the all-zero input, but the branch predictor performs similarly.
- In the **regular** input set, half of the bits are set to one in a regular pattern. This is a repeating pattern of eight bits followed by eight bits set to zero. This implies that the conditional code is executed in half of the iterations.
- In the **random** input set, half of the bits are set to one as well, but now the pattern of zeroes and ones is generated by a pseudo-random generator. Consequently, this input will result in the same amount of code executed as for the regular input set, but branch prediction will be much less accurate, resulting in more branch misses and higher execution times.

Together, these four input sets allow us to study to what extent our proposed transformations are able to eliminate timing dependencies that originate from different control flow for different keys or from branch prediction behavior that depends on keys.

Please note that the number of times each loop was invoked per experiment differs for the three microbenchmarks. For each benchmark, the number of invocations was chosen to be a good balance between short experimentation times and accurate measurements.

Finally we applied our approach to a function from the OpenSSL ³

³<http://www.openssl.org>

library, version 0.9.8i, that can be used to implement RSA. We studied in particular the execution time of the function `BN_sub`. This library code is executed on two different inputs that result in different timing behavior in the original code. These inputs were obtained by running the original code on multiple random-generated inputs, where we observed how many times each code path was taken during the execution. For one input, only one code path was executed, while for the other input both code paths were executed equally.

We ran all experiments on an Intel Core 2 Duo machine running Linux at 2.2GHz. All versions were executed 20 times on all inputs to collect statistics on the timing behavior and branch prediction. All binary code was generated using LLVM 2.3's standard compiler options to generate 64-bit code, except for the OpenSSL code. For the OpenSSL code we disabled some compiler middle-end optimizations because they resulted in partially if-converted code even when using the unadapted compiler. Since we are measuring the impact of applying if-conversion, we use code without any if-conversion as a baseline. These numbers represent an upper limit on the overhead.

4.4.2 Register-based dependencies

Our approach transforms control dependencies into data dependencies. The effectiveness of our approach depends on whether or not these data dependencies can introduce variations in the execution time themselves. Since our compiler back-end uses conditional move instructions to remove the key-dependent branches from a program, we investigated whether or not the use of conditional move instructions can introduce variable execution times.

Consider the following loop bodies:

```

1 body1:  mov    ecx, edi
2         add    eax, ebx
3
4 body2:  mov    eax, edi
5         add    eax, ebx
6
7 body3:  cmp     edx, 0
8         cmov   eax, edi
9         add    eax, ebx

```

In the first two loop bodies, the second instruction adds the value in register `ebx` to the value in register `eax`. If the first loop body is exe-

cuted in a loop, all additions in subsequent iterations are dependent on each other: register `eax` serves as a kind of accumulator, to which the value in `ebx` is added repeatedly. In the second loop body, each iteration starts with a fresh value being copied into `eax`. The register renaming pipeline stages in out-of-order processors is able to detect this. When we measured the execution of an unrolled loop of loop body 2 on an Intel Core 2 Duo, it was 40% faster than the execution of an unrolled loop of body 1.

The third loop body can, depending on the condition flag, do the same computation as the first or second loop body. The `cmov` instruction is a conditional move instruction. When the *zero flag* is set to true, `cmov` will copy the value of register `edi` into register `eax`, otherwise the value of register `eax` remains unchanged. The zero flag is set by comparing the value of register `edx` to zero. Thus, the computation performed by loop body 3 will be the same as either body 1 or 2, depending on the value of `edx`. Since loop bodies 1 and 2 had different timing behavior, the question is whether the value of `edx` can influence the timing behavior of loop body 3.

On an Intel Core 2 Duo, we measured the execution time of this loop body in an unrolled loop for different values of `edx`, i.e., for the case in which the conditional move is executed, and for the case in which it is not executed. No significant timing differences were observed, indicating that the answer to the above question is no. So whereas loop body 1 and loop body 2 had different dependencies between instructions, resulting in different timing behavior, the value of `edx` in loop body 3 has no influence on the timing behavior.

The reason is that x86 processors implement the conditional move instructions as an instruction that is executed unconditionally.⁴ This instruction has two source operands, one of which is implicitly the same as the destination operand. From these two source operands, one is selected based on the condition flags, and then written to the destination operand. This can be implemented at the hardware level by a multiplexer.

Furthermore, we conjecture that this analysis holds for all processors. For the x86 processors and ARM processors we know of, the analysis holds. The reason is that these out-of-order processors rely on the

⁴We did not find public information on this subject, but the correctness of our findings and our assumption on the implementation of the conditional move instructions was confirmed privately by Intel engineers.

reordering of operations to achieve high performance. Fundamentally, these processors try to execute instructions as soon as they can. This means that the processor will consider an instruction ready for execution as soon as it can somehow determine that there are no more true data dependencies that require the instruction to wait for the result of other instructions still being executed. The technique used to differentiate true data dependencies from false data dependencies is called register renaming [136]. Register renaming is simplified significantly if the processor already knows in the processor pipeline front-end which instructions depend on which other instructions. So register renaming is simplified by implementing a conditional move instruction by means of a multiplexer instruction. The drawback of this implementation is that even when the processor somehow knows beforehand that the condition flag is false, the processor cannot exploit this information to get rid of superfluous dependencies. Furthermore, this multiplexer requires an additional register read per conditional move instruction. On most processors this drawback is not big enough to warrant the implementation of more complex register renaming techniques. So it is safe to assume that on x86 processors and on many other out-of-order processors, such as the ARM Cortex-A9, conditional execution will not introduce data-dependent timing behavior as long as the conditional execution is limited to move instructions.

We verified this experimentally on both Intel and ARM processors. For Intel, we verified this on the Core 2 Duo processors, and for ARM we verified it on the in-order Cortex-A8 core, and the out-of-order Cortex-A-9 and Cortex A-15 cores⁵.

4.4.3 Effectiveness

Table 4.1 presents average execution times and average branch mispredictions, as well as other statistics on the various modular exponentiation microbenchmarks and on the OpenSSL benchmark for the different inputs.

Table 4.1(a) presents, for each microbenchmark version and for the appropriate input set, the execution times averaged over 20 runs. Table 4.1(b) presents the standard deviation of the measured times. Clearly, for all of the benchmarks, the original versions behave quite

⁵For the Cortex-A9, ARM engineers confirmed our theoretical analysis in private communication.

differently for their different inputs. Hence, these benchmark versions leak information about the secret inputs.

The if-converted benchmarks have significantly longer execution times, because of the overhead introduced by the if-conversion, but the execution times obtained with the different inputs now display significantly more similarity. The same can be observed in the benchmarks after if-conversion and division elimination, indicating that our approach has indeed removed all timing dependence on the secret inputs.

To assess the confidence with which we can draw the above conclusion, we have performed multiple t-tests [63]. Table 4.1(c) depicts the p-values obtained from t-tests applied to three combinations of inputs: combination all zero - all one, combination regular - random, and combination input1 - input2.

For the 32-bit and 64-bit modular exponentiation, we see that if-conversion alone does not suffice to achieve a high confidence. This follows from the presence of a variable-latency division instruction. For the same microbenchmarks, we then emulated the division instruction with a call to a constant-time division instruction, as described in Section 4.3. Table 4.1 also contains measurements with emulated division instructions. From those measurements, we can conclude with confidence that our approach does in fact result in key-independent timing behavior.

For the 256-bit modular exponentiation, if-conversion alone proves to be sufficient to eliminate all key-dependent timing behavior. In this 256-bit big integer implementation, the divisor of all executed division instructions is the fixed 64-bit value `0x00000000ffffffff`. The compiler replaces this division by a constant with a multiplication and a shift. As a result, no division instructions occur in the compiled code.

For the OpenSSL code, the same reasoning holds: there are no division instructions, so we only need to apply if-conversion to make this code's timing behavior independent of the input key.

Tables 4.1(d), 4.1(e) and 4.1(f) present similar statistical information for the number of branch mispredictions measured with performance counters. Similar conclusions can be drawn. Here as well, we can be confident that if-conversion and division instruction elimination are successful in eliminating timing behavior dependencies on the secret inputs. Still, two remarks need to be made here.

Firstly, very high standard deviations were obtained in the numbers of measured branch mispredictions with several versions of mod-

exp32. We repeated these experiments multiple times, but never obtained more consistent results. We can not explain why the standard deviations of 1193.2 and 514.8 are as high as they are. Clearly, however, these high standard deviations do not occur in the if-converted code. Lower standard deviations are important to us because they allow us to draw stronger conclusions about the transformed programs.

Secondly, the confidence scores 0.0077 and 0.2701 for the number of branch mispredicts of the if-converted `modexp32` and `modexp64` benchmarks are relatively low, notwithstanding the fact that there is no key-dependent control flow present in those benchmarks. The explanation for this result can presumably be found in the complex pipeline behavior of the Intel Core 2 Duo pipeline. Remember that there still are variable-latency division instructions present in these benchmark versions. The actual latencies of those instructions occurring during the execution of the program have an effect on the number of branches that are fetched by the processor and issued speculatively [136]. So while the number of truly executed branches (so-called retired branches) is independent of the secret inputs of these benchmarks, the number of speculatively issued branches is not. Hence the number of mispredicted branches, some of which were speculative, is not independent either. While we cannot verify this behavior due to the closed nature of Intel's processor, this is a reasonable explanation, given the measurements and an abstract knowledge on how out-of-order processor architectures work. If division instructions are eliminated as well, this effect does not play anymore, and then we can conclude with much higher confidence that the number of branch mispredictions does not depend on the secret inputs anymore.

4.4.4 Efficiency

Figure 4.5 displays the average performance overhead of applying if-conversion and, where necessary, the elimination of division instructions. We averaged this overhead over a large number of pseudorandom inputs. Since the performance overhead will depend on the original execution time, which depends heavily on the input, we have opted to show the average overhead of the transformation on pseudorandom inputs. Because the LLVM middle-end itself already if-converted the code in *f2*, and because the code in *f1* only features a single execution path, these fragments undergo no additional transformations in our approach. So we observe no slowdown for them.

Table 4.1: Statistical results of if-conversion and the elimination of variable-latency division instructions

| | original | | | | if-converted | | | | if-converted + div elimination | | | |
|-----------|----------|---------|---------|--------|--------------|---------|---------|--------|--------------------------------|---------|---------|--------|
| | all zero | all one | regular | random | all zero | all one | regular | random | all zero | all one | regular | random |
| modexp32 | 0.785 | 1.377 | 1.027 | 1.223 | 1.504 | 1.535 | 1.524 | 1.515 | 26.473 | 26.473 | 26.474 | 26.474 |
| modexp64 | 0.911 | 1.816 | 1.354 | 1.405 | 1.847 | 1.897 | 1.871 | 1.877 | 21.109 | 21.110 | 21.109 | 21.109 |
| modexp256 | 3.642 | 7.374 | 5.532 | 5.531 | 16.764 | 16.757 | 16.759 | 16.760 | | | | |
| | | | | | | | | | | | | |
| | input 1 | | input 2 | | input 1 | | input 2 | | | | | |
| OpenSSL | 1.580 | | 2.255 | | 4.606 | | 4.606 | | | | | |

(a) average execution times (in seconds)

| | original | | | | if-converted | | | | if-converted + div elimination | | | |
|-----------|----------|---------|---------|--------|--------------|---------|---------|--------|--------------------------------|---------|---------|--------|
| | all zero | all one | regular | random | all zero | all one | regular | random | all zero | all one | regular | random |
| modexp32 | 0.015 | 0.000 | 0.001 | 0.003 | 0.001 | 0.001 | 0.001 | 0.001 | 0.007 | 0.007 | 0.007 | 0.007 |
| modexp64 | 0.000 | 0.001 | 0.000 | 0.000 | 0.000 | 0.001 | 0.000 | 0.001 | 0.005 | 0.005 | 0.005 | 0.004 |
| modexp256 | 0.009 | 0.011 | 0.004 | 0.007 | 0.029 | 0.018 | 0.007 | 0.007 | | | | |
| | | | | | | | | | | | | |
| | input 1 | | input 2 | | input 1 | | input 2 | | | | | |
| OpenSSL | 0.060 | | 0.013 | | 0.015 | | 0.025 | | | | | |

(b) standard deviation of execution times

| | original | | if-converted | | if-converted + div elimination | |
|-----------|--------------------|----------------|--------------------|----------------|--------------------------------|----------------|
| | all zero - all one | regular-random | all zero - all one | regular-random | all zero - all one | regular-random |
| modexp32 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.8525 | 0.9557 |
| modexp64 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.6028 | 0.9644 |
| modexp256 | 0.0000 | 0.0000 | 0.3539 | 0.6781 | | |
| | | | | | | |
| | input 1 - input 2 | | input 1 - input 2 | | | |
| OpenSSL | 0.0000 | | 0.9405 | | | |

(c) p-value of the t-test applied to the execution times

| | original | | | | if-converted | | | | if-converted + div elimination | | | |
|-----------|----------|---------|---------|--------|--------------|---------|---------|--------|--------------------------------|---------|---------|--------|
| | all zero | all one | regular | random | all zero | all one | regular | random | all zero | all one | regular | random |
| modexp32 | 8698 | 11 | 2539 | 44043 | 11 | 11 | 11 | 11 | 91 | 91 | 91 | 91 |
| modexp64 | 1173 | 16 | 2866 | 17230 | 507 | 507 | 507 | 507 | 65510 | 65510 | 65509 | 65509 |
| modexp256 | 1155 | 25521 | 19249 | 19123 | 270 | 269 | 269 | 267 | | | | |
| | | | | | | | | | | | | |
| | input 1 | | input 2 | | input 1 | | input 2 | | | | | |
| OpenSSL | 14 | | 55311 | | 17 | | 17 | | | | | |

(d) average number of branch mispredictions (x1000)

| | original | | | | if-converted | | | | if-converted + div elimination | | | |
|-----------|----------|---------|---------|--------|--------------|---------|---------|--------|--------------------------------|---------|---------|--------|
| | all zero | all one | regular | random | all zero | all one | regular | random | all zero | all one | regular | random |
| modexp32 | 1193.2 | 0.1 | 57.1 | 514.8 | 0.1 | 0.1 | 0.1 | 0.2 | 0.6 | 0.5 | 0.7 | 0.8 |
| modexp64 | 58.4 | 0.2 | 7.1 | 11.2 | 1.2 | 1.1 | 1.0 | 1.1 | 0.8 | 0.5 | 0.3 | 0.4 |
| modexp256 | 38.5 | 11.7 | 14.1 | 19.9 | 8.0 | 7.3 | 8.7 | 9.4 | | | | |
| | | | | | | | | | | | | |
| | input 1 | | input 2 | | input 1 | | input 2 | | | | | |
| OpenSSL | 0.1 | | 1401.7 | | 0.3 | | 0.3 | | | | | |

(e) standard deviation of the number branch mispredictions (x1000)

| | original | | if-converted | | if-converted + div elimination | |
|-----------|--------------------|----------------|--------------------|----------------|--------------------------------|----------------|
| | all zero - all one | regular-random | all zero - all one | regular-random | all zero - all one | regular-random |
| modexp32 | 0.0000 | 0.0000 | 0.0077 | 0.3370 | 0.9487 | 0.8205 |
| modexp64 | 0.0000 | 0.0000 | 0.2701 | 0.6406 | 0.9916 | 0.7508 |
| modexp256 | 0.0000 | 0.6194 | 0.6909 | 0.6106 | | |
| | | | | | | |
| | input 1 - input 2 | | input 1 - input 2 | | | |
| OpenSSL | 0.0000 | | 0.3569 | | | |

(f) p-value of the t-test applied to the number of branch mispredictions

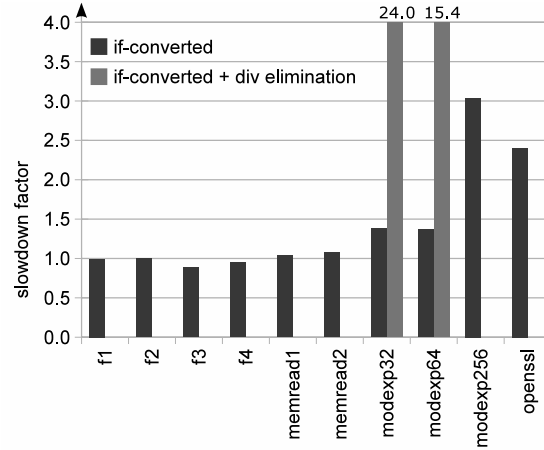


Figure 4.5: Average (over pseudo-random inputs) execution slowdown after applying if-conversion and elimination of variable-latency division instructions.

Surprisingly, the simple functions $f3$ and $f4$ became faster after if-conversion. This can be attributed to the improved branch prediction. As there are less branches to predict in the converted code, and in particular less branches that depend on pseudo-random inputs, less cycles are lost after mispredictions.

For the other microbenchmarks, increasing complexity results in additional overhead. The maximum overhead observed corresponds to a slowdown with a factor 24.0. This is very large, but it will only occur in those program fragments that (1) involve computations on sensitive keys, and (2) include division instructions.

4.4.5 Code size overhead

The increase in code size due to if-conversion and elimination of divisions is depicted in Figure 4.6. As is to be expected, our control experiment, $f1$, has kept the same size, since the original function already had straight-line control flow. There is also no increase in code size for experiment $f2$, because the compiler middle-end had already applied if-conversion. The other experiments also behave as expected: the more complicated the control flow, the greater the increase in code size after if-conversion. Similarly, the more memory operations there are, the more the code size increases. Without the elimination of variable-

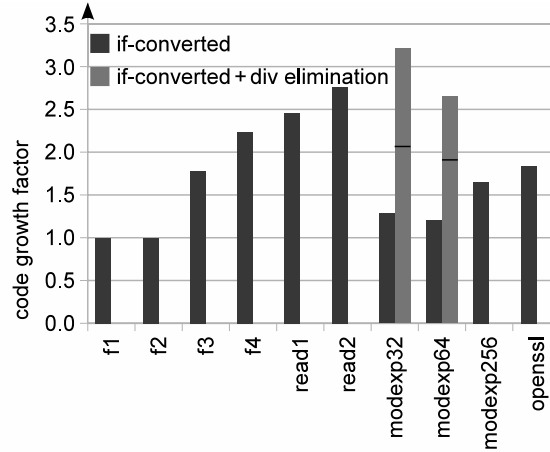


Figure 4.6: Code size increase after applying if-conversion and elimination of variable latency division instructions. The horizontal bars indicate the additional overhead due to the constant-time division function.

latency division instructions, the code becomes up to 2.75 times larger. When we also replace the division instructions by calls to a subroutine, the code sizes become up to 3.21 times larger.

To assess these results correctly, two observations need to be made. Firstly, in real applications, these code size increases will only occur on those fragments that need to be protected because they involve key-dependent control flow. The remaining part of the code will remain the same, so the overall code size increase will be much more limited. Secondly, in the code sizes after division instruction elimination, the size of the subroutines that implement the division operations are included. Their relative contribution to the total code size is marked with the horizontal line in the bars in Figure 4.6: the fraction above the line comes from the newly included division subroutines. Their absolute sizes are 77 bytes for the 32-bit version, and 80 bytes for the 64-bit version. Since programs are typically much larger than 80 bytes, this clearly indicates that their contribution to the code size increase in real applications will be minimal.

4.5 Comparison with existing techniques

Several approaches have been proposed before to mitigate side channels in cryptographic code. In what follows, we discuss some existing

techniques and compare them to our approach.

4.5.1 Source-based solutions

One class of solutions is to rewrite the source code so that the source code's control flow no longer depends on the secret information. Molnar et al. [108] presented an approach relying on source-to-source transformations. Because conditional execution is not available on all architectures, Molnar et al. presented a variation on our use of conditional execution. Consider the following C code fragment, which is very similar to a fragment in their paper:

```

1  if (n != 0) {
2      if (n % 2) {
3          r = r * b;
4          n = n - 1;
5      } else {
6          b = b * b;
7          n = n / u;
8      }
9  }
```

They rewrite this fragment as

```

1  m1 = -(n != 0);
2  m2 = m1 & -(n % 2);
3  r = (m2 & (r * b)) | (~m2 & r);
4  n = (m2 & (n - 1)) | (~m2 & n);
5  m2 = m1 & ~m2;
6  b = (m2 & (b * b)) | (~m2 & b);
7  n = (m2 & (n / u)) | (~m2 & n);
```

Rather than using conditional execution, Molnar et al. use bit masking. Conditions are used to generate masks `m1` and `m2` that consists of all zeros or all ones, and then they mimic the conditional execution by using the masks. As is obvious from the example, this allows them to handle nested conditions. However, there are very fundamental issues with their approach, that we will discuss next.

Compilers As Molnar et al. indicate themselves, some compilers will translate the negation operator into conditional branches. They found this to be the case for the very popular GCC compiler, and for that reason they could not use that compiler. Likewise, they found it necessary

to write a static verifier that can validate whether the compiled code really provides key-independent control flow. This weakens their claim that a source-to-source approach is practical.

This also hints at another problem of their approach: if the compiler is able to see through their masking constructs, he will be able to optimize the code to, for example, get rid of unnecessary copy operations involving temporary variables or to propagate constant values through the program [110]. Furthermore, it is possible that the compiler will be able to optimize the code by reintroducing conditional branches. In other words, if one uses a compiler that is smart enough to optimize the rewritten code, one risks that the compiler is so smart that it will optimize the code too much, from a security point of view. And if the compiler is not smart enough to see through the masking, he will simply perform no optimizations.

By contrast, in our compiler back-end approach relying on conditional execution, the compiler has already optimized the code using the full potential of the original code that was much easier to analyze. As such, full compiler optimization and guaranteed key-independent control flow can be combined without any problem.

Exceptions and side-effects Molnar et al. treat side effects such as exceptions and store instructions incorrectly. They claim that they can neglect exceptions because they only consider correct programs in the first place, in which no exceptions occur. This is incorrect. In the above code fragments, in the original code the division would only have been executed if $(n \neq 0)$ and if $(n \% 2)$ was zero. Hence, in the original version, it does not matter what the value of u is if those conditions are not met. So it is perfectly fine to execute the original fragment with, for example, n being 5, and u being 0. No division-by-zero exception will occur. In the rewritten code however, the exception will occur, as the division is executed unconditionally in that code. This clearly changes the program behavior, which is unacceptable for a program transformation.

Finally, Molnar et al. do not discuss how to handle conditional function calls, and conditional loads or stores. This is not a fundamental issue of their approach, it is merely an incompleteness. Similar techniques as the one we use to safe-guard instructions can be used in their approach.

4.5.2 Binary rewriting

Other compiler-based techniques have been proposed as well. Bayrak et al. [22] use compiler transformations to mitigate certain power-based attacks. They first analyze the execution of the program, determine which instructions introduce variation in the power traces, and then selectively apply countermeasures in their compiler extension. Their approach is complementary to ours: while we do not consider any other side channels than time, they do not consider any other side channels than power.

4.5.3 Hardware instructions

Hardware extensions can offer secure and efficient solutions for dealing with side channels in cryptographic code. Because they are efficient, they should be used whenever possible, rather than our approach. However, such extensions are limited in two ways.

Only some algorithms are supported When processor designers add cryptographic extensions, they do not add support for all possible cryptographic algorithms. This means that implementers of algorithms that cannot be expressed in terms of the secure extensions, will still have to rely on software-based mitigation techniques such as ours.

Long time before general availability Designing, testing, and making processors takes time. Thus, there can be a significant amount of time between the standardization of a cryptographic algorithm, and the availability of the relevant hardware extensions. For example, the Rijndael cipher was standardized as AES in 2001 [115]. The first cache timing attacks against AES were published in 2005 [23, 119]. Still, it was only in 2010 that Intel released its first processor to include support for AES primitives [74]. Even then, not all customers of Intel processors will immediately switch to such processors. Until all customers have switched to processors with hardware mitigations, software-based mitigation techniques can still prove valuable.

Our side-channel aware compiler can include knowledge of existing cryptographic hardware instructions, and generate code that uses them if they are available, but can fall back to slower code when the used

extensions are not available on the processor on which the generated code runs.

4.6 Conclusion

We introduced a timing side-channel aware compiler. It allows developers to automatically protect their applications against timing side channels. We have showned how if-conversion can be applied to remove control flow that depends on private information from a program. Furthermore, we introduced different solutions for two data-flow dependent timing leaks. We we also discussed the overhead associated with these different mitigation strategies.

Chapter 5

Conclusions & Future work

In this PhD work, we investigated the influence of different kinds of program variation on the security of programs. In this chapter, we summarize the conclusions obtained in the course of this work, and add some further thoughts. Of course, no work is ever finished, which is why we will also discuss some potential future research directions.

5.1 Conclusions

5.1.1 Variation between program versions

Real-world attack tools can be used effectively by attackers to recover source-code induced mutations in programs by comparing an unpatched binary with a patched binary. We introduced models that can be used to automate the behavior of an attacker using real-world attack tools. Thus, we can automatically approximate the effort required by an attacker using such tools.

We used real attack tools to show that an attacker can indeed easily find the source-code induced mutations from a binary patch. The efforts required by the attacker can be significantly increased by introducing artificial variation between the original and patched versions of a program.

Furthermore, we have shown that we can include the output from attack tools to introduce these artificial changes only in the locations where it will increase the attack effort, while keeping the overhead low. We iteratively use feedback from attack tools to choose which diversification transformations to apply to which code fragments. This itera-

tive, feedback-driven approach was implemented and evaluated in our Glaucus diversification framework.

This research shows that software vendors that wish to protect their patches from immediate exploitation can use software diversity to increase the attack effort. However, there are some choices to be made in order to apply software diversity. In fact, both attackers and software vendors will have to make different trade-offs.

Software vendors will have to consider the urgency of the vulnerability. When a white-hat hacker contacts a developer about a zero-day vulnerability for which he has an exploit, it can be critical to fix the vulnerability without exposing it publicly by means of an all too obvious patch. When some bug has been known publicly for a long time and no exploit has ever been constructed, there will be little need to protect a patch. Similarly, this urgency will also depend on the speed with which users apply updates: software that automatically updates itself with security patches will have a faster adoption rate than software that users have to re-install from scratch.

Software vendors will also need to decide how much overhead they will tolerate. This is both overhead in execution time as overhead in patch size. Too much overhead in execution time might be unacceptable, except in the most urgent cases. In cases of an urgent patch, the software vendor could decide that a higher overhead is acceptable until most users have upgraded, and then release a separate, undiversified update without any performance overhead. Similarly, software diversity can introduce a significant overhead in binary patch size. It has to, since small patches will immediately point the attacker to the changed instructions. However, this will also increase the bandwidth needed to distribute the patch.

While we have shown how to represent these trade-offs, we cannot decide on them ourselves. They will depend on the operational and economic situation of the software vendor, in addition to the importance of the security issue that is being fixed.

Similarly, the attacker will have a trade-off to make as well. By focusing on small subsets of the attack tool's output, the attacker can reduce his attack time. However, this comes at a cost of reducing the amount of SCIMs he will be able to identify. When no SCIMs remain in the set of changes the attacker studies manually, he will of course have no success in finding the vulnerability. Either the attacker then abandons his search, or starts again by considering a less constraining set of

heuristics. While we can represent the attacker's options, we can only do so with the additional knowledge of the ground truth. The quality of the different attack heuristics depends on the SCIMs. The attacker's only a priori knowledge of SCIMs is contained in the description of the security fix. Whether or not this description is enough to correctly decide the best heuristics will depend on the specific situation. Thus, while the software vendor has all the information needed to make an informed decision of which strategy to use in an attack, the attacker possibly himself starts without any required information to choose his best option.

Once enough users have applied the patch, there is no longer a need for the details of the SCIMs to remain protected. It is then up to the software vendor to make a trade-off between the time the SCIMs are protected and how much time users have to apply the patch. In fact, a similar trade-off exists in the open source community, where it can be argued that security patches should be developed into a private repository and only put into a public repository at a later time to hide the details of security patches before they are released to the public [19].

5.1.2 Variation of execution time in a program

Execution time variation can leak a significant amount of information. While the execution time will depend both on data flow and control flow, this PhD work focused on removing the execution time variation caused by control flow, although we also investigated the effects of some data-dependent operations. This work resulted in a timing side-channel aware compiler, which can generate binary code without timing variation. Thus, we have shown that a software vendor can effectively protect himself against attacks where the execution time variation depends on the control flow using this compiler.

However, rewriting the control flow dependencies into data flow dependencies can introduce significant execution time overhead. Still, software vendors want their programs to be as fast as possible. In this respect, cryptographic software does not differ from other software. The ideal scenario would obviously for all cryptographic code to have private data-independent control flow by design. Then all a timing side-channel aware compiler would have to do, is to ensure that it does not introduce any new control flow dependencies through any of its optimization passes. Sadly, this is not the case. Thus, software vendors

have a trade-off between cost and overhead. At the extreme points of this trade-off are rewriting all cryptographic code manually to make it free of timing side channels, or automatically rewriting all code with a timing side-channel aware compiler. While manually rewriting code is the most expensive solution, it can also result in the lowest overhead. Similarly, the automatic solution will have the lowest cost, but may incur more overhead. A middle ground could be to rewrite the most computation-intensive code manually (or even to allow some variation to still exist there), while automatically rewriting the remaining code. How much code can be automatically protected will again depend on the specific use case.

Our side-channel aware compiler will also remove some data-dependent timing variation. It removes the timing variation by replacing some variable-latency instructions with custom fixed-latency instruction sequences. This again introduces a significant execution time overhead. However, since a compiler is able to target specific architectures, a side-channel aware compiler might have knowledge of which architectures actually exhibit the timing-dependent behavior. Similarly, when some architectures have explicit support to disable timing-dependent behavior, the compiler can have knowledge of this as well. Thus, when the compiler generates code for a specific architecture, it can generate code that is optimized for speed on this particular architecture, while still being side-channel free.

5.2 Future work

5.2.1 Variation between program versions

Both attacks using program variation and defenses against such attacks can be improved, based on our current work:

- An attacker could try to automatically detect which transformations have been applied to a binary, and undo or normalize their effects before comparing the programs. Depending on how easily an attacker can normalize existing diversification transformations, more resilient transformations could be investigated.

Currently we have a master thesis student working on implementing and evaluating normalization transformations. The transformations all use dynamic information. The first trans-

formation is to remove all unexecuted code, and rewriting all remaining branch instructions so that all their edges point to executed code, by replacing conditional branches with unconditional ones if necessary. Dynamically computed jump targets are replaced by their dynamically observed values. Similarly, calls to branch functions are replaced by the control transfers that are dynamically observed. Preliminary results show a significant improvement in the number of matched instructions.

- While the feedback to select specific transformations for Glaucus currently only consists of the heuristics BinDiff used, it would be worthwhile to also include feedback from other diffing tools. In particular, including feedback on the heuristics used by Binary-Differ would be worthwhile.
- While the cost function for Glaucus only uses execution count information, we could also include information about the code size overhead that is introduced. This should result in binaries with lower code size overhead. Furthermore, we could also use a better estimation of the performance overhead.
- We could also try to decrease the apparent program variation as seen by the different attack tools. One potential technique would rewrite the programs so that they are emulators of a virtual instruction set. The actual program code is then rewritten in this virtual instruction set and added to the program as data. Any SCIM is then in the data section, which would remain undetected by most diffing tools.
- Another possibility is to create a different kind of model for patch-based attacks. Our current approach evaluates actual attack tools. However, it might be possible to try and model the underlying assumption all these tools make, and create an abstract model for all similar diffing tools. This model can then be used in combination with a certified compiler to generate code that comes with both a proof that the generated code is equivalent to the source code, and a proof that the generated code is resilient against the modeled attacks. Such models have already been proposed for obfuscation [73] and have been integrated into the CompCert C compiler [25].

5.2.2 Variation of execution time in a program

We introduced a timing side-channel aware compiler. We can extend this work in multiple ways:

- Rewriting control dependencies into data dependencies can significantly increase the overhead. We remove all key-dependent control dependencies in the program. However, this need not be quite so extreme. We could just as well only selectively remove control dependencies. As described in the related work, work already exists that can estimate the information leakage caused by timing variation. This technique could be combined with our side-channel aware compiler, so that it initially only removes the control flow that leaks the most information. Then, depending on how much overhead is considered acceptable, more control flow could be removed, ordered by its information leakage.
- Our approach rewrites all code that depends on private information, which introduces a slowdown. We could combine our techniques with a just-in-time compiler that by default executes the fast but time-dependent code. When it detects a potential attack, it switches at run time to the slower but timing-independent code. This line of research is currently being investigated by a PhD student at our group.
- While our compiler is timing side-channel aware, other existing work has introduced a power side-channel aware compiler [22]. Both approaches could be combined into a single side-channel aware compiler. Such a compiler would probably have to target architectures more specifically than we do, since the possible power leakages and their mitigations will depend more on the features of the target architecture than our timing mitigations do.
- Our work focused on removing timing side channels. However, it is not always clear if and how timing variation can be exploited by an attacker. Some large time variations might not give useful information to an attacker, while some small timing variations do. An interesting line of research would be to automate timing side-channel attacks as well. We could model the execution time of a program on a processor core, and use this model to automate an attack that extracts the private information.

Bibliography

- [1] Onur Aciçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *ASIACCS '07*, pages 312–320, 2007.
- [2] Onur Aciçmez. Yet another microarchitectural attack: exploiting I-cache. In *Proceedings of the First Computer Security Architecture Workshop (CSAW)*, pages 11–18, 2007.
- [3] Onur Aciçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 110–124, 2010.
- [4] Onur Aciçmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 80–91. IEEE, 2007.
- [5] Dakshi Agrawal, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi. The EM sidechannel (s). *Cryptographic Hardware and Embedded Systems-CHES 2002*, pages 29–45, 2003.
- [6] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 177–189, 1983.
- [7] Bertrand Anckaert. *Diversity for Software Protection*. PhD thesis, Ghent University, 2008.
- [8] Bertrand Anckaert, Mariusz Jakubowski, and Ramarathnam Venkatesan. Proteus: virtualization for diversified tamper-resistance. In *Proceedings of the workshop on Digital Rights Management*, pages 47–58, 2006.

- [9] Bertrand Anckaert, Matias Madou, and Koen De Bosschere. A model for self-modifying code. In *Proceedings of the 8th Information Hiding Conference*, volume 4437 of *LNCS*, pages 232–248, 2007.
- [10] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. Graph-based malware detection using dynamic analysis. *Journal in Computer Virology*, 7:247–258, 2011.
- [11] Geneviève Arboit. A method for watermarking Java programs via opaque predicates. In *The Fifth International Conference on Electronic Commerce Research (ICECR-5)*, 2002.
- [12] ARM and Thumb-2 instruction set quick reference card. http://infocenter.arm.com/help/topic/com.arm.doc.qrc00011/QRC0001_UAL.pdf.
- [13] ARM architecture reference manual ARMv7-A and ARMv7-R edition, 2010.
- [14] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS)*, pages 297–307. ACM, 2010.
- [15] David I. August, John W. Sias, Jean-Michel Puiatti, Scott A. Mahlke, Daniel A. Connors, Kevin M. Crozier, and Wen mei W. Hwu. The program decision logic approach to predicated execution. In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 208–219, Washington, DC, USA, 1999. IEEE Computer Society.
- [16] Algirdas Avizienis and Liming Chen. On the implementation of N-version programming for software fault tolerance during execution. In *The 1st IEEE Computer Software and Applications Conference*, pages 149–155, 1977.
- [17] Michael Backes and Boris Köpf. Formally bounding the side-channel leakage in unknown-message attacks. *Computer Security-ESORICS 2008*, pages 517–532, 2008.
- [18] Elena Gabriela Barrantes, David Ackley, Stephanie Forrest, and Darko Stefanovi. Randomized instruction set emulation. *ACM Trans. on Info. and Syst. Secu.*, 8(1):3–40, 2005.

- [19] Adam Barth, Saung Li, Benjamin Rubinstein, and Dawn Song. How open should open source be? arXiv:1109.0507v1, 2011.
- [20] Ulrich Bayer, Engin Kirda, and Christopher Kruegel. Improving the efficiency of dynamic malware analysis. In *Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10*, pages 1871–1878, New York, NY, USA, 2010. ACM.
- [21] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. TTAanalyze: A tool for analyzing malware. In *15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, 2006.
- [22] Ali Galip Bayrak, Francesco Regazzoni, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. A first step towards automatic application of power analysis countermeasures. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 230–235. IEEE, 2011.
- [23] Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, The University of Illinois at Chicago, 2005.
- [24] Sandeep Bhatkar, Daniel DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *The 12th USENIX Security Symposium*, pages 105–120, 2003.
- [25] Sandrine Blazy and Roberto Giacobazzi. Towards a formally verified obfuscating compiler. In *SSP 2012-2nd ACM SIGPLAN Software Security and Protection Workshop*, 2012.
- [26] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *Cryptographic Hardware and Embedded Systems CHES 2006, LNCS volume 4249*, pages 201–215. Springer, 2006.
- [27] Gunnar Brinkmann and Bart Coppens. An efficient algorithm for the generation of planar polycyclic hydrocarbons with a given boundary. *Match-Communications in Mathematical and in Computer Chemistry*, 62(1):209–220, 2009.
- [28] Julien Brouchier, Tom Kean, Carol Marsh, and David Naccache. Temperature attacks. *Security & Privacy, IEEE*, 7(2):79–82, 2009.
- [29] Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. *Advances in Cryptology-ASIACRYPT 2009*, pages 667–684, 2009.

- [30] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. *Computer Security–ESORICS 2011*, pages 355–371, 2011.
- [31] David Brumley and Dan Boneh. Remote timing attacks are practical. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, 2003.
- [32] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *IEEE Symposium on Security and Privacy*, pages 143–157, 2008.
- [33] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Code normalization for self-mutating malware. *Security & Privacy, IEEE*, 5(2):46–54, 2007.
- [34] Mariano Ceccato, Massimiliano Di Penta, Jasvir Nagra, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. Towards experimental evaluation of code obfuscation techniques. In *Proceedings of the 4th ACM workshop on Quality of protection*, pages 39–46. ACM, 2008.
- [35] Mariano Ceccato, Massimiliano Di Penta, Jasvir Nagra, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. The effectiveness of source code obfuscation: an experimental assessment. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 178–187. IEEE, 2009.
- [36] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. White-box cryptography and an AES implementation. In *Selected Areas in Cryptography*, pages 250–270. Springer, 2003.
- [37] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. A white-box DES implementation for drm applications. *Digital Rights Management*, pages 1–15, 2003.
- [38] M. Christodorescu, S. Jha, J. Kinder, S. Katzenbeisser, and H. Veith. Software transformations to improve malware detection. *Journal in Computer Virology*, 3(4):253–265, 2007.
- [39] The Clang Team. *Clang Compiler User's Manual*, 2013.

- [40] Frederick Cohen. Operating system evolution through program evolution. *Computers and Security*, 12(6):565–584, 1993.
- [41] J. Coke, H. Balig, N. Cooray, E. Gamsaragan, P. Smith, K. Yoon, J. Abel, and A. Valles. Improvements in the Intel Core 2 processor family architecture and microarchitecture. *Intel Technology Journal*, 12(03):179–192, 2008.
- [42] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th Conference on Principles of Programming Languages*, pages 184–196. ACM Press, 1998.
- [43] Bart Coppens, Bjorn De Sutter, and Koen De Bosschere. Protecting your software releases. *IEEE Security & Privacy*, 11(2):47–54, 2013.
- [44] Bart Coppens, Bjorn De Sutter, and Jonas Maebe. Feedback-driven binary code diversification. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):24:1–24:26, 2013.
- [45] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 45–60. IEEE Computer Society, 2009.
- [46] Core Security Technologies. Windows SMTP service DNS query id vulnerabilities. CoreLabs Security Advisory, 2010.
- [47] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *The 15th USENIX Security Symposium*, pages 105–120, 2006.
- [48] Anthony Cozzie, Frank Stratton, and Samuel T. King Hui Xue. Digging for data structures. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.
- [49] Scott A. Crosby, Dan S. Wallach, and Rudolf H. Riedi. Opportunities and limits of remote timing attacks. *ACM Transactions on Information and System Security (TISSEC)*, 12(3):17, 2009.

- [50] Bruno De Bus, Dominique Chanet, Bjorn De Sutter, Ludo Van Put, and Koen De Bosschere. The design and implementation of FIT: a flexible instrumentation toolkit. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 29–34. ACM, 2004.
- [51] Bruno De Bus, Bjorn De Sutter, Ludo Van Put, Dominique Chanet, and Koen De Bosschere. Link-time optimization of ARM binaries. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 211–220, 2004.
- [52] Bjorn De Sutter, Bertrand Anckaert, Jens Geiregat, Dominique Chanet, and Koen De Bosschere. Instruction set limitation in support of software diversity. In *11th International Conference on Information Security and Cryptology ICISC 2008*, number 5461 in LNCS, pages 152–165, 12 2008.
- [53] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. Sifting out the mud: Low level C++ code reuse. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 275–291, 2002.
- [54] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Trans. on Prog. Lang. and Syst.*, 27(5):882–945, 9 2005.
- [55] Saumya Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compression. In *Workshop on Compiler Support for System Software*, pages 378–415, 1999.
- [56] Daniel Dolz and Gerardo Parra. Using exception handling to build opaque predicates in intermediate code obfuscation techniques. *Journal of Computer Science & Technology*, 8(2), 2008.
- [57] Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications*, pages 1–3, 2005.
- [58] Stefan Dziembowski and Krzysztof Pietrzak. Leakage-resilient cryptography. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 293–302. IEEE, 2008.
- [59] Chris Eagle. *The IDA Pro Book*. No Starch Press, 2nd edition, 2011.

- [60] Nicolás Economou. Microsoft Virtual PC: The hyper-hole-visor bug & MS10-048: Win32k window creation vulnerability (CVE-2010-1897), 2010.
- [61] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32.Stuxnet dossier, version 1.4. Technical report, Symantec, 2011.
- [62] Peter Feiner, Angela Demke Brown, and Ashvin Goel. Comprehensive kernel instrumentation via dynamic binary translation. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 135–146. ACM, 2012.
- [63] Thomas S. Ferguson. *Mathematical Statistics: A Decision Theoretic Approach*. Academic Press, 1967.
- [64] Halvar Flake. Structural comparison of executable objects. In *Proceedings of the Detection of Intrusions and Malware & Vulnerability Assessment, GI SIG SIDAR Workshop*, pages 161–173, 2004.
- [65] Christophe Foket, Bjorn De Sutter, Bart Coppens, and Koen De Bosschere. A novel obfuscation: Class hierarchy flattening. In *Proceedings of 5th International Symposium on Foundations and Practice of Security*, 2012.
- [66] Stephanie Forrest, Anil Somayaji, and David Ackley. Building diverse computer systems. In *The Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.
- [67] Michael Franz. E unibus pluram: massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 workshop on New security paradigms*, pages 7–16. ACM, 2010.
- [68] Free Software Foundation, Inc. *GNU Compiler Collection (GCC) Manual*, 2013.
- [69] Stefan Frei, Thomas Duebendorfer, and Bernhard Plattner. Firefox (in) security update dynamics exposed. *ACM SIGCOMM Computer Communication Review*, 39(1):16–22, 2008.
- [70] Jeroen Frijters. Reverse engineering the MS10-060 .NET security patch. Blogpost, 2010.
- [71] Steven D. Galbraith. *Mathematics of Public Key Cryptography*. Cambridge University Press, 2012.

- [72] Debin Gao, Michael K. Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *Proceedings of the 10th International Conference on Information and Communications Security, ICICS '08*, pages 238–255, Berlin, Heidelberg, 2008. Springer-Verlag.
- [73] Roberto Giacobazzi, Neil D Jones, and Isabella Mastroeni. Obfuscation by partial evaluation of distorted interpreters. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, pages 63–72. ACM, 2012.
- [74] Shay Gueron. Advanced encryption standard (AES) instructions set. Technical report, Intel Mobility Group, 2008.
- [75] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games—bringing access-based cache attacks on aes to practice. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 490–505. IEEE, 2011.
- [76] Shon Harris, Allen Harper, Chris Eagle, and Jonathan Ness. *Gray hat hacking: the ethical hacker's handbook*. McGraw-Hill, 2008.
- [77] Hans Hoogstraaten, Ronald Prins, Daniël Niggebrugge, Danny Heppener, Frank Groenewegen, Janna Wettinck, Kevin Strooy, Pascal Arends, Paul Pols, Robbert Kouprie, Steffen Moorrees, Xander van Pelt, and Yun Zheng Hu. Black Tulip – report of the investigation into the DigiNotar Certificate Authority breach. Technical report, Fox-IT, 2012.
- [78] Nick Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *Designs, Codes and Cryptography*, 23(3):283–290, 2001.
- [79] W.M. Hu. Reducing timing channels with fuzzy time. *Journal of Computer Security*, 1(3):233–254, 1992.
- [80] Robert Hundt, Easwaran Raman, Martin Thureson, and Neil Vachharajani. Mao – an extensible micro-architectural optimizer. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '11*, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.
- [81] Intel. *IA-32 Intel Architecture Software Developer's Manual*, 2003.

- [82] Intel. *IA-32 Intel Architecture System Programming Guide*, 2003.
- [83] Nephi Johnson. From patch to proof-of-concept: MS10-081. Blog-post, 2011.
- [84] Gaurav Kc, Angelos Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *The 10th ACM Conference on Computer and Communications Security*, pages 272–280, 2003.
- [85] Alexander Klink and Julian ‘Zeri’. Effective denial of service attacks against web application platforms. 28th Chaos Communication Congress.
- [86] Paul Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology - CRYPTO96*, pages 104–113. Springer, 1996.
- [87] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology - CRYPTO 99, LNCS 1666*, pages 388–397. Springer-Verlag, 1999.
- [88] Boris Köpf and David A. Basin. An information-theoretic model for adaptive side-channel attacks. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 286–296. ACM, 2007.
- [89] Boris Köpf and Markus Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *Computer Security Foundations Symposium, 2009. CSF’09. 22nd IEEE*, pages 324–335. IEEE, 2009.
- [90] David G. Korn, Joshua P. MacDonald, Jeffrey C. Mogul, and Kiem-Phong Vo. Request for comments: 3284, the VCDIFF generic differencing and compression data format. Technical report, The Internet Society, 2002.
- [91] Christopher Krügel, William K. Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium*, pages 255–270, 2004.

- [92] Arun Lakhotia, Mila Dalla Preda, and Roberto Giacobazzi. Fast location of similar code fragments using semantic ‘juice’. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, page 5. ACM, 2013.
- [93] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A taxonomy of computer program security flaws, with examples. *ACM Computing Surveys (CSUR)*, 26(3):211–254, 1994.
- [94] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Tech. Report UIUCDCS-R-2003-2380, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Sep 2003.
- [95] Nate Lawson and Taylor Nelson. Exploiting timing attacks in widespread systems. Black Hat USA Briefings, 2019.
- [96] Byoungyoung Lee and YeongJin Jang. Exploit shop website, 2012.
- [97] Clifford Liem. Tools and Methodologies for Layered, Diverse, and Renewable Security in Tethered Systems. In *Second Int. Workshop on Remote Entrusting*, 2009.
- [98] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and Communications Security*, pages 290–299, 2003.
- [99] Mark Loveless. Corporate security: A hacker perspective, 2006.
- [100] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.
- [101] Matias Madou, Bertrand Anckaert, Bjorn De Sutter, and Koen De Bosschere. Hybrid static-dynamic attacks against software protection mechanisms. In *Proceedings of the 5th ACM workshop on Digital Rights Management*, pages 75–82, 2005.

- [102] Jonas Maebe, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere. Javana: a system for building customized Java program analysis tools. In *ACM SIGPLAN Notices*, volume 41, pages 153–168. ACM, 2006.
- [103] Jonas Maebe and Koen De Bosschere. Instrumenting self-modifying code. *Workshop on Automated and Algorithmic Debugging*, pages 103–113, 2003.
- [104] Jonas Maebe, Michiel Ronsse, and Koen De Bosschere. Diota: Dynamic instrumentation, optimization and transformation of applications. In *Compendium of Workshops and Tutorials held in conjunction with PACT02*, 2002.
- [105] Bourquin Martial, King Andy, and Robbins Edward. Binslayer: Accurate comparison of binary executables. In *2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW 2013)*, 2013.
- [106] Robert Martin, John Demme, and Simha Sethumadhavan. Time-warp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 118–129. IEEE Press, 2012.
- [107] Barton P. Miller and Kevin A. Roundy. Binary-code obfuscations in prevalent packer tools. *ACM Computing Surveys*, June 2012.
- [108] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks, 2005.
- [109] HD Moore. Exploiting IIS via HTMLEncode (MS08-006). Blog-post, 2008.
- [110] Steve Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- [111] Vijayanand Nagarajan, Xiangyu Zhang, Rajiv Gupta, Matias Madou, Bjorn De Sutter, and Koen De Bosschere. Matching control flow of program versions. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance*, pages 83–94, 2007.

- [112] Danny Nebenzahl, Shmuel Sagiv, and Avishai Wool. Install-time vaccination of windows executables to defend against stack smashing attacks. *Dependable and Secure Computing, IEEE Transactions on*, 3(1):78–90, 2006.
- [113] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007.
- [114] Adam O'Donnell and Harish Sethu. On achieving software diversity for improved network security using distributed coloring algorithms. In *Proceedings of the 11th ACM conference on Computer and Communications Security*, pages 121–131. ACM Press, 2004.
- [115] United States National Institute of Standards and Technology. Federal information processing standards publication 197. Technical report, NIST, 2001.
- [116] Jeongwook Oh. Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries. In *BlackHat USA*, 2009.
- [117] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers Track at the RSA Conference 2006*, pages 1–20. Springer-Verlag, 2006.
- [118] Colin Percival. Naive differences of executable code. <http://www.daemonology.net/bsdifff/>, 2003.
- [119] Colin Percival. Cache missing for fun and profit. *BSDCan 2005*, 2005.
- [120] Igor Shparlinski Phong Q. Nguyen. The insecurity of the digital signature algorithm with partially known nonces. *Journal of Cryptology*, 15(3):151–176, 2002.
- [121] Igor V Popov, Saumya K Debray, and Gregory R Andrews. Binary obfuscation using signals. In *USENIX Security Symposium*, pages 275–290, 2007.
- [122] The Chromium Projects. Software updates: Courgette. Technical report, Google, 2010.
- [123] Andre Protas and Steve Manzuik. Skeletons in Microsoft's closet - silently fixed vulnerabilities. *BlackHat Europe*, 2006.

- [124] Daan Raman, Bjorn De Sutter, Bart Coppens, Stijn Volckaert, Koen De Bosschere, Pieter Danhieux, and Erik Van Buggenhout. DNS tunneling for network penetration. In *Annual International Conference on Information Security and Cryptology*, 2012.
- [125] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009.
- [126] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
- [127] Bruno P. S. Rocha, Sruthi Bandhakavi, Jerry den Hartog, William H. Winsborough, and Sandro Etalle. Towards static flow-based declassification for legacy and untrusted programs. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 93–108. IEEE, 2010.
- [128] Michiel Ronsse, Bastiaan Stougie, Jonas Maebe, Frank Cornelis, and Koen De Bosschere. An efficient data race detector backend for diota. *Advances in Parallel Computing*, 13:39–46, 2004.
- [129] Nathan E. Rosenblum, Xiaojin Zhu, Barton P. Miller, and Karen Hunt. Learning to analyze binary computer code. *Artificial Intelligence (AAAI), Chicago, IL*, pages 798–804, 2008.
- [130] Daniel Y. Deng Ruirui C. Huang and and G. Edward Suh. Orthrus: Efficient software integrity protection on multi-cores. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 371–384. ACM, 2010.
- [131] Todd Sabin. Comparing binaries with graph isomorphisms. Technical report, BindView RAZOR Team, 2004.
- [132] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 33–46. ACM, 2009.

- [133] Michael Schlansker, Scott Mahlke, and Richard Johnson. Control cpr: a branch height reduction optimization for epic architectures. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 155–168, New York, NY, USA, 1999. ACM.
- [134] Justin Seitz. *Gray Hat Python*. No Starch Press, 2009.
- [135] Monirul Sharif, Andrea Lanzi, Jonathon Griffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In *IEEE Symposium on Security and Privacy*, pages 94–109, 2009.
- [136] John Shen and Mikko Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill, 2005.
- [137] Stelios Sidiroglou, Sotiris Ioannidis, and Angelos D. Keromytis. Band-aid patching. In *Proceedings of the 3rd workshop on Hot Topics in System Dependability, HotDep'07*, Berkeley, CA, USA, 2007. USENIX Association.
- [138] Kevin Skadron, Pritpal S Ahuja, Margaret Martonosi, and Douglas W Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 259–271. IEEE Computer Society Press, 1998.
- [139] Slawlerguy. Reversing the ms08-067 patch... Blogpost, 2008.
- [140] David Solomon. *Data Compression: The Complete Reference*. Springer, 2007.
- [141] Alexander Sotirov. Reverse engineering Microsoft binaries. CanSecWest, 2006.
- [142] Iain Sutherland, George E. Kalb, Andrew Blyth, and Gaius Mulley. An empirical examination of the reverse engineering process for binary files. *Computers & Security*, 25(3):221–228, 2006.
- [143] Jeroen Van Cleemput, Bart Coppens, and Bjorn De Sutter. Compiler mitigations for time attacks on modern x86 processors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):23, 2012.
- [144] Jan-Willem van de Waerdt. *The TM3270 Media-processor*. PhD thesis, Technische Universiteit Delft, 2006.

- [145] Nibin Varghese. Reverse engineering for exploit writers. Clubhack, 2008.
- [146] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in Xen. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 41–46. ACM, 2011.
- [147] Andrew Walenstein, Rachit Mathur, Mohamed R. Chouchane, and Arun Lakhotia. Constructing malware normalizers using term rewriting. *Journal in Computer Virology*, 4(4):307–322, 2008.
- [148] Harsimran Walia. Reversing Microsoft patches to reveal vulnerable code. Nullcon, 2011.
- [149] Chenxi Wang, Jack Davidson, Jonathan Hill, and John Knight. Protection of software-based survivability mechanisms. In *Proceedings of the 2nd International Conference of Dependable Systems and Networks*, pages 193–202. IEEE Computer Society Press, 2001.
- [150] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia, 12 2000.
- [151] Zheng Wang, Ken Pierce, and Scott McFarling. Bmat – a binary matching tools for stale profile propagation. *The Journal of Instruction-Level Parallelism*, 2:1–20, 2000.
- [152] Zhenghong Wang and Ruby B. Lee. Covert and side channels due to processor architecture. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 473–482, Washington, DC, USA, 2006. IEEE Computer Society.
- [153] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. *SIGARCH Comput. Archit. News*, 35(2):494–505, 2007.
- [154] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *19th ACM Conference on Computer and Communications Security (CCS)*, pages 157–168, 2012.

- [155] Xiangyu Zhang and Rajiv Gupta. Matching execution histories of program versions. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 197–206, 2005.
- [156] Xiangyu Zhang and Rajiv Gupta. Whole execution traces and their applications. *ACM Trans. on Architecture and Code Optimization*, 2(3):301–334, 2005.
- [157] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316. ACM, 2012.
- [158] Yongxin Zhou and Alec Main. Diversity via code transformations: A solution for NGNA renewable security. In *NCTA - The National Show*, 2006.
- [159] Zynamics. *Zynamics BinDiff Manual*, 2012.